

INRIA Lille Nord Europe
LIFL UMR CNRS 8022

PhD Thesis in Computer Science
2011-2014

Heterogeneity and Locality -Aware Work Stealing for Large Scale Branch-and-Bound Irregular Algorithms

Trong-Tuan Vu

Directors:

Nouredine Melab, Professor, University of Lille 1, France
Bilel Derbel, Associate Professor, University of Lille 1, France

Reviewers: Pierre Sens - Professor, University of Paris 6, France
Daniel Tuyttens - Professor, University of Mons, Belgium

Abstract

Branch and Bound (B&B) algorithms are exact methods used to solve combinatorial optimization problems (COPs). The computation process of B&B is extremely time-intensive when solving large problem instances since the algorithm must explore a very large space which can be viewed as a highly irregular tree. Consequently, B&B algorithms are usually parallelized on large scale distributed computing environments in order to speedup their execution time. Large scale distributed computing environments, such as Grids and Clouds, can provide a huge amount of computing resources so that very large B&B instances can be tackled. However achieving high performance is very challenging mainly because of (i) the irregular characteristics of B&B workload and (ii) the heterogeneity exposed by large scale computing environments.

This thesis addresses and deals with the above issues in order to design high performance parallel B&B on large scale heterogeneous computing environments. We focus on dynamic load balancing techniques which are to guarantee that no computing resources are underloaded or overloaded during execution time. We also show how to tackle the irregularity of B&B while running on different computing environments, and consider to compare our proposed solutions with the state-of-the-art algorithms. In particular, we propose several dynamic load balancing algorithms for homogeneous, node-heterogeneous and link-heterogeneous computing platforms.

Firstly, in homogeneous computing environments, we propose to organize computing resources following a logical peer-to-peer overlay and to distribute the load according to the so-defined overlay [Vu 2012]. Large scale experiments involving up to 1200 computing cores are reported and the performance of our approach in terms of deployment cost, parallel efficiency, speed-up, and scalability, is finely analyzed. Compared to previous approaches of B&B such as Master-Worker or Hierarchical Master-Worker, our approach provides significant improvements. Besides, we investigate the interaction between work distribution and the properties of the overlay topology, and study their cumulative impact on the performance of the whole system. These results provide new insights in understanding how topology and work-sharing are crucial to design efficient and scalable distributed systems.

Secondly, in node-heterogeneous environments with possibly multiple cores, multiple CPUs and multiple GPU devices, we describe two approaches addressing the critical issue of how to map B&B workload with the different levels of parallelism exposed by the target compute platform. We essentially deal with two issues: (i) the difference of computing power when a mixture of multiple distributed CPUs and multiple GPUs devices [Vu 2013] is considered, and (ii) when multiple shared memory cores are additionally considered [Vu 2014b]. We thereby contribute a throughout large scale experimental study which allows us to derive a comprehensive and fair analysis of the proposed approaches under different system configurations using up to 20 GPUs and up to 512 distributed cores. In particular, we are able to

obtain linear speed-ups at moderate scales where adaptive load balancing among the heterogeneous compute resources is shown to have a significant impact on performance. At the largest scales, intra-node and inter-node parallelism are shown to have a crucial importance in order to alleviate locking issues among shared memory threads and to scale the distributed resources while optimizing communication costs and minimizing idle times.

Finally, in link-heterogeneous computing environments, we investigate the design of a generic dynamic load balancing algorithm [Vu 2014a] which can be easily implemented to fit different types of link-heterogeneity. The proposed algorithm extends on reference approaches, namely Probabilistic Work Stealing (PWS), and Adaptive Cluster-aware Random Stealing (ACRS); by introducing new adaptive control operations that are shown to be highly accurate in increasing work locality and decreasing steals cost. We further conduct an extensive experiment with many link-heterogeneity configurations ranging from a Grid scenario with two-level communication hierarchy, to Peer-to-Peer systems with a more complex communication hierarchy. Over all experimented configurations, our results show that although the proposed protocol is not tailored for a specific networked platform, it can save 30% execution time compared to its competitors, while demonstrating high quality self-adjusting capabilities.

Keywords:

Parallel Branch-and-Bound, Dynamic Load Balancing, Work Stealing, Heterogeneous Computing, Combinatorial Optimization, Exact Methods, Grid'5000.

Contents

Introduction

The PhD Thesis, presented in this document, deals with the important topic of improving the performance of *Large-scale Combinatorial Optimization* on *Heterogenous Distributed Computing Environments*. It has been prepared within the DOLPHIN research group from CNRS/LIFL, Inria Lille-Nord Europe and Université Lille 1. We also note that this thesis was funded through the Inria HEMERA project.

Combinatorial Optimization Problems (COPs)[1] arise in many application domains including all the areas of technology and industry management. Solving them is a process of finding one or several best solutions(s) in a set of finite feasible solutions. There are two main types of resolution methods for COPs: heuristics and exact methods. Heuristics produce high quality solutions in a reasonable time, but no guarantee is given on the optimality of computed solutions. Conversely, exact methods allow to find the optimal solutions with the proof of optimality, but the computation time can be very huge in particular for large problem instances. Among the exact methods, Branch and Bound (B&B) algorithms are the most commonly used to solve COPs in practice. Instead of searching the optimal solutions exhaustively, B&B algorithms wisely discard some branches which do not contain the optimal solution, hence reducing the search space and computation time. However, although such a technique allows to reduce effectively the search space generated by the corresponding instances, it still remains unsatisfactory to solve large instances within a reasonable time. As a result, the use of parallelism allows us to improve the resolution time while dealing with large instances. In this thesis, we deal with the design of parallel B&B algorithms while using paradigms coming from distributed and high performance computing (HPC).

In the HPC field, since the last decades we have witnessed an impressive evolution of computing technologies. A typical solution is to use supercomputers equipped with hundreds thousands of processing cores connected by a local high-speed computer network. One can find a list of the most powerful supercomputers in the world at the TOP500 [Top500]. Though providing an impressive computing power, supercomputers are very expensive so that only big research institutes and industrial enterprises are able to fund such supercomputers. With the arise of high speed networks, computing resources are nowadays available in the form of clouds, grids, aggregated clusters and personal computers geographically distributed around the world. Such computing resources provide a great computing power which is in theory sufficient to solve all the large instances of COPs. However in practice achieving this power is very challenging due to many challenging issues. Among them, heterogeneity is one of the most crucial factor with respect to the overall performance of distributed computing systems. The heterogeneity in computation and communication is a natural aspect of distributed large scale systems as different compute nodes

[1] An optimization problem can be formulated as a minimization or maximization of a cost function. Without loss of generality, we consider minimization problems in this document.

of different architectures are connected via different networks of different communication cost. Dealing with the heterogeneity is indeed a key challenge to explore the potential computing power of such heterogeneous computing resources.

Parallel B&B algorithms on distributed computing systems appear to be a useful tool to solve the large instances of COPs. During the resolution process, parallel B&B yields compute irregularity and raises difficult challenges. In the course of the execution, task units are generated dynamically which are unknown beforehand, hence causing unbalanced workload among processing units. This issue requires further communications in order to balance workload among processing units in overloaded and underloaded areas. In particular, most of recent works [Mezmaz 2007a, Bendjoudi 2012a, Chakroun 2013a] are only based on the Master-Worker or Hierarchical Master-Worker paradigm which are strongly limited regarding scalability in the large scale computing environments. To overcome this issue, some approaches [Djamai 2013] are proposed based on the Peer-To-Peer paradigm. Nevertheless, to the best of our knowledge, these existing approaches still do not focus on the irregularity issues of parallel B&B and the heterogeneity aspects of distributed computing systems.

The contribution of this thesis is to deal with such issues at the aim of improving the current state-of-the-art parallel B&B approaches. In this dissertation, we propose new generic guidelines for building efficient parallel B&B. Specifically, we target the three main questions that arise when designing parallel B&B in large scale distributed computing environments:

1. Which is a right paradigm for solving the irregularity issues or the unbalanced workload of parallel B&B?

2. What is the impact of the computation's heterogeneity on the performance of parallel B&B?

3. What is the impact of the communication's heterogeneity on the performance of parallel B&B?

To answer the first question, we study some dynamic load balancing algorithms which dynamically solve the unbalanced world situations at runtime. Among them, we found that the work stealing algorithm [Burton 1981, Blumofe 1999] might be a suitable solution for the irregularity problems. In contrast with previous works of parallel B&B [Mezmaz 2007a, Bendjoudi 2012a, Djamai 2013, Chakroun 2013a], the work stealing approach is fully distributed and does not rely on any central point in making decisions for balancing workload. Let us notice that despite a rich references of parallel B&B in the literature [Mezmaz 2007a, Bendjoudi 2012a, Djamai 2013, Chakroun 2013a], the work stealing paradigm has not been considered in order to solve the workload irregularity of parallel B&B while running on distributed computing environments. We further proposed a tree-based overlay load balancing algorithm which uses the tree structure in making load balancing decisions. Our algorithm is novel in the sense that it allows all the processing units cooperate to each other according to the tree structure while balancing the workload.

We experimentally show that both approaches can significantly improve speedups of parallel B&B compared to other algorithms [Mezmaz 2007a, Bendjoudi 2012a].

To answer the second question, we consider the common characteristics of distributed computing systems where processing units are heterogeneous and their potential computing power might be different in order of magnitude. For instance, a processing unit can be a single core CPU, a multi-core GPU or a CPU equipped with GPU devices, etc. To deal with this type of heterogeneity, we propose to estimate the compute capability of processing units continuously at runtime with respect to the problem instance being executed so that tasks can be accordingly assigned to all processing units by load balancing operations. Moreover we present an efficient load balancing mechanism for Multi-cores systems where load balancing operations takes the hardware hierarchy (intra-node *vs* inter-node) into consideration.

To answer the third question, we consider the current geographically distributed computing platforms where processing units are distributed such as Grids, Peer-To-Peer or Global Computing. In this context, all the processing units communicate to each other via different network links with different communication latencies. In particular, we investigate how link-heterogeneity of such distributed platforms impacts the performance of parallel B&B. We also propose an efficient and generic approach to deal with dynamic load balancing for link-heterogeneous computing platforms. The proposed approach is generic in the sense that it can be deployed in different distributed computing contexts exposing different properties in terms of communication latency.

Thesis Structure

Chapter 1

In the first chapter, we will describe general concepts of parallel Branch and Bound (B&B) and its challenges. For this purpose, we will first detail the sequential Branch and Bound algorithms. Thereafter, we will present some common parallel models of B&B. We will then present the architecture of shared and distributed memory computing systems and highlight different challenges that appear when dealing with parallel B&B. The rest of the chapter is dedicated to the description of several existing works of parallel distributed B&B.

Chapter 2

In the second chapter, we will describe in details work stealing algorithms and their design for parallel distributed B&B. Thereby, we will further present tree-based dynamic load balancing algorithms. Furthermore, extensive experiments up to the scale of 1200 processing units demonstrate the efficiency of the tree-based algorithms as well as the strong potential of work stealing compared to Master-Worker or Hierarchical Master-Worker paradigm.

Chapter 3

In the third chapter, the focus is on the efficient control of node-heterogeneity when designing dynamic balancing algorithms for parallel distributed B&B. We will first introduce our approach, called 2MBB (Multi-CPUs Multi-GPUs Parallel B&B), while considering a completely distributed settings where all processing units are distributively connected via a network and each processing unit is either a single CPU-core or a single CPU-core equipped with a GPU device. We will then present the 3MBB (Multi-cores Multi-CPUs Multi-GPUs Parallel B&B) as the Multi-core processors appear in heterogeneous platforms. Finally, we will present the extensive experiments up to 512 CPU cores and 20 GPUs in order to show the performance of these approaches.

Chapter 4

In the fourth chapter, we will mainly investigate the performance of parallel B&B while running on geographically distributed link-heterogeneous computing systems. In particular, we consider the Grid with two-level communication hierarchy and the Peer-To-Peer with more complex communication hierarchy. Different state-of-the-art work stealing algorithms including our generic approaches will be presented and experimented on different complicated link-heterogeneous settings.

Chapter 1

Parallel Branch and Bound on Distributed System

Contents

1.1 Introduction

Many real-world problems can be modeled as Combinatorial Optimization Problems (COPs). They are NP-hard and CPU-time intensive, therefore solving them to optimality requires a huge amount of computing resources. Generally speaking, solving COPs consists in searching in a very large space comprising all possible solutions in order to find one or several best solution(s). Instead of the exhaustive enumeration of all possible solutions of COPs, the Branch and Bound (B&B) algorithm performs an implicit enumeration and properly eliminates some parts which will not eventually lead to the optimal solutions. In other words, B&B is a technique that allows to find the optimal solution by keeping the best solution found so far. If a new (partial) solution is not able to improve the current best one, it will be eliminated for reducing the size of the search space. Although the elimination mechanism of B&B can be effective, the search space generated by large instances of COPs is still too huge to be fully explored by the sequential B&B in a reasonable time. From the perspective of HPC community, computing resources are available in the form of aggregated clusters, grids and personal computers scattered over possibly large scale distributed platforms connected via networks. These types of resources provide many levels of parallelism to improve the computing performances of B&B while solving large instances in parallel.

In this chapter, we introduce all the key concepts at the crossroads of B&B and Distributed Computing which are closely related to our works. The chapter is organized as follows. Section 1.2 describes the B&B algorithms. Section 1.3 presents some classifications of parallel B&B in the literature. Section 1.4 provides possible implementation designs for parallel B&B. Section 1.5 presents the characteristics of the considered computing environments ranging from shared to distributed memory system. Section 1.6 briefly highlights some related works in parallel B&B on the considered systems. Finally, section 1.7 concludes the chapter.

1.2 Serial Branch and Bound

Branch and Bound (B&B) algorithm is a well-known technique for solving to optimality NP-hard optimization problems. It implicitly enumerates all the possible solutions over a search space and looks for the global optimal one. The search space is constructed at runtime as a tree whose root node is the original problem. The intermediate nodes of the tree, which are obtained by decomposing from their parents, represent corresponding subproblems and can be recursively decomposed into other subproblems in next iterations. The leaves represent potential solutions of the original problem which can not be decomposed further. The main idea of B&B algorithms is to detect and discard all nodes which do not contain or lead to the global optimal solution. The optimization process stops when all nodes are explored or discarded. A B&B algorithm contains the following four basic operations: **branching**, **bounding**, **selection**, and **pruning**.

- **Branching**: the branching operation divides a (parent) subproblem into two or more (child) subproblems depending on the level of the parent subproblem in the B&B tree. Then the generated child subproblems will be taken into account in the next iteration. Generally, this operation decomposes a given problem (which is difficult to solve directly) into smaller problems (which are easier to solve) based on some constraints. In a B&B tree, only the root or the intermediate nodes are processed by this operation. These nodes are recursively branched/decomposed until reaching the leaves.

- **Bounding**: the bounding operation estimates a bound value for all generated subproblems according to the objective of the considered instance. The bound value of a subproblem then allows the algorithm to decide whether or not it is necessary to explore this subproblem.

- **Selection**: the selection operation determines a strategy to select the next subproblems of a B&B tree to be processed. There are three basic strategies for B&B algorithms: depth-first, bread-first and best-first. Depth-First B&B functions like a DFS algorithm in graph theory which aims to explore as far as possible along each branch of a B&B tree until reaching a leaf. Hence a node with deepest level in the tree will be chosen for exploration in the next iteration. The main idea of this strategy is to allow the algorithm to find a new solution quickly. Similarly, in Best-First B&B strategy, a node with the best bound so far will be selected to process. It aims to look for a good solution rapidly so that the algorithm is able to eliminate as many remaining subproblems in the tree as possible. In practice, the performance of this strategy depends on the considered instance. In contrast, the Breadth-first B&B performs like a BFS algorithm which explores all nodes of the same level before moving to the next one. For instance, when a problem is decomposed into several subproblems, all the subproblems are fully explored before moving to process the new generated nodes. Therefore the number of nodes at each level of the tree grows exponentially with the level making it infeasible to solve large problems.

- **Pruning**: the pruning operation reduces the size of the tree. In a B&B tree, it eliminates all branches which surely can not lead to the best solution. Technically speaking, this operation compares the current best found solution with a bound value of a subproblem and eliminates the subproblem if its bound value is worse than the current best found solution. For instance, in a minimization problem, if a lower bound of a subproblem is greater than the current best found solution, then the subproblem is discarded since we will not find any *better* solutions in the subspace rooted at the subproblem. It is worth to notice that eliminating a subproblem allows to implicitly prune the subspace rooted at this subproblem.

Algorithm 1 describes a sequential template of a typical B&B algorithm. The algorithm terminates when all subproblems are explicitly or implicitly visited.

Algorithm 1: A Template of sequential B&B Algorithm

```
Data: r: root node;
Optimal_Solution: the optimal solution of the problem r
Optimal_Value: the value of the optimal solution of the problem r
Found_Solution: the current best found solution of the problem r
Found_Value: the value of the current best found solution of the problem r
Q: queue of B&B subproblems
1  Q ⟵ r ;
2  while Q.size > 0 do
3      P ⟵ SELECT() ;
4      if P is LEAF then
5          P.cost ⟵ f(P) ;
6          if P.cost < Found_Value then
7              Found_Value ⟵ P.cost ;
8              Found_Solution ⟵ P ;
9      else
10         P.lowerbound ⟵ f(P) ;
11         if P.lowerbound < Found_Value then
12             (P_1, ..., P_k) ⟵ BRANCH(P) ;
13             Q ⟵ (P_1, ..., P_k) ;
14         else
15             PRUNE(P)
16     Optimal_Value ⟵ Found_Value ;
17     Optimal_Solution ⟵ Found_Solution ;
```

1.3 Parallel Branch and Bound algorithm

1.3.1 Classification of Parallel B&B

Parallel B&B algorithms have attracted a lot of attention from the community. Many references can be found in the literature [Trienekens 1986, Trienekens 1992, Gendron 1994, Bourbeau 2000, Melab 2005]. Most of them identify and investigate the potential parallelism in designing parallel B&B. The potential source of parallelism comes from the characteristics of the B&B algorithm as well as from the considered computing environments. In the following, we briefly discuss some existing classifications of parallel B&B.

1.3.1.1 Classification of Trienekens et al.

Trienekens *et al.* [Trienekens 1986, Trienekens 1992] classified parallel B&B into low and high level according to their degree of parallelization:

- Low level: Some parts of the sequential B&B are parallelized, and the others

are processed sequentially. But these parts are interacted as similar as the original sequential one. For instance, the branching, bounding, selection or pruning operation can be processed in parallel by several threads/processes. Therefore the overall behavior of this approach resembles the behavior of the sequential B&B algorithm (e.g they process the same subproblem at a given order during the execution)

- High Level: In case of a high level parallelization, the effects and consequences of the parallelism introduced are not restricted to a particular part of the B&B, but influence the algorithm as a whole. The work performed by the parallel algorithm does not need to be equal to the work performed by the sequential algorithm. The order in which the work is performed can differ, and it is even possible that some parts of the work performed by the parallel algorithm are not performed by the sequential algorithm, or *vice versa*. For example, several iterations of the main loop can be performed in parallel (e.g., several processes executing the algorithm branch in parallel from their own subproblem).

1.3.1.2 Classification of Gendron *et al.*

Gendron *et al.* [Bourbeau 2000, Gendron 1994] identified three types of parallel B&Bs according to the degree of parallelism of the search tree:

- Parallelism of type 1 (node-based): introduces parallelism when performing the operations on generated subproblems. For instance, it consists in executing the bounding operation in parallel for each subproblem to accelerate the execution. Thus, this type of parallelism has no impact on the general structure of the B&B algorithm and is particular to the problem to be solved. Clearly, this type is similar to the Low level of the classification of Trienekens *et al.*

- Parallelism of type 2 (tree-based): consists in building the solution tree in parallel by performing operations on several subproblems simultaneously. Hence, this type of parallelism may affect the design of the algorithm. This type of parallelization is suitable for coarse-grained asynchronous MIMD architectures. This type is also similar to the High Level classified by Trienekens *et al.*

- Parallelism of type 3 (multi-search): implies that several solution trees are generated in parallel. The trees are characterized by different operations (branching, bounding, and pruning), and the information generated when building one tree can be used for the construction of another.

1.3.1.3 Classification of Melab

The most recent classification is the one done by Melab in [Melab 2005] and reported by Mezmaz in [Mezmaz 2007a]. In this taxonomy, four models of parallel B&B

algorithms are identified: parallel multi-parametric model, parallel tree exploration model, parallel evaluation of the bounds, and parallel evaluation of a single bound. They are essentially a mix of the previously described classifications.

- Parallel multi-parametric model: consists in considering several coarse-grained B&B algorithms. Each algorithm uses its own parameters and several variants of an algorithm can be obtained by modifying these parameters. Therefore, different B&B algorithms can be obtained and executed in parallel. This type is not commonly used in practice.

- Parallel tree exploration model: this model is similar to the type 2 classified by Gendron *et al.* or the high level model classified by Trienekens *et al.* In this model, different subtrees can be explored in parallel. Therefore, the branching, bounding, selection, and elimination can be executed simultaneously

- Parallel evaluation of the bounds: this model is similar to the type 1 classified by Gendron *et al.* or the low level model classified by Trienekens et al. It allows a parallel evaluation of the subproblems generated by the branching operator. This model is exploitable only if the bounds evaluation is entirely executed after the branching operation. However, it is not suitable for grid environments. Indeed, in the case of synchronous model, additional delays are engendered because of the heterogeneous nature of grid resources.

- Parallel evaluation of a single bound: This model does not change the semantics of the algorithm because it identical to the sequential version but the evaluation phase is faster. This model depends on the considered problem and it is in general synchronous and centralized. It is limited in terms of extensibility, nevertheless, it is efficient when combined with other models.

1.4 Implementation considerations

The above section presents some different classifications of parallel B&B in the literature. These studies mainly investigated potential parallelism sources of B&B and proposed possible models to parallelize B&B on the current architectures of computing systems. Among the proposed models, the parallel tree exploration model, which has been studied in several research on parallel B&B algorithms [Casado 2008, Mezmaz 2007a, Djamai 2011b], was shown to be the most suitable for high performance and scalability. In fact, it is the most coarse-grain and often implemented in MIMD architectures. In this type of parallelism, each processing unit performs the exploration to simultaneously process different parts of a B&B tree [Casado 2008, Mezmaz 2007a, Djamai 2011b]. Technically speaking, there are two important issues when designing and implementing a parallel B&B: **work pool management** and **communication model**.

1.4.1 Work pool management

In parallel B&B, work pool management plays an important role in the performance of the algorithm. In fact, at each iteration of B&B, one has to perform two operations related to work pool. Algorithm 1 shows that, an iteration of a B&B consists in a procedure to pop a subproblem from a pool, perform a set of operations (branching, bounding and pruning), then insert one or several generated subproblem(s) into the same pool. Generally speaking, a work pool is simply a data structure (e.g. array, list, map, stack, queue, etc) placed in a memory location where processing units can find and store the generated subproblems during execution. In other words, work pool management handles the I/O of B&B. In parallel B&B, simultaneous I/O operations of several processing units to the same work pool is critical for performance which poses several challenges of how to handle them effectively. In practice, there are two common strategies for managing work pool:

- **Single pool** algorithms use one single memory location. They are implemented both on shared and distributed memory systems. In a shared memory system, a single global work pool is shared among worker threads. All threads concurrently get work units (e.g generated B&B subproblems) from the global single work pool. When they finish their processing, the threads push new generated work units to the global work pool. Synchronization techniques are unavoidable to synchronize among worker threads when popping/pushing a node from/to the single global pool. In distributed memory system, single pool algorithms are implemented using a master-slave paradigm. Typically, the master sends work units to its slaves and the slaves send back results (i.e new generated works) to the master. The single pool provides a global picture of unexplored and pending works in the whole system. Therefore, this makes it easy to design, implement and deploy. But, a single pool is subject to a bottleneck limiting its scalability since only one process can access to the pool at a given time.

- **Multiple pool** algorithms simply use multiple memory locations to store work units. There exist some variants depending on the number of pools used in the system. The three most popular are collegial, grouped and mixed. In the first case, each processing unit has its own private pool. In the second one, all processing units are partitioned in several groups and each group shares the same work pool. The choice of the number of pools depends on the number of processing units as well as their accessing frequency. The last one is a mix between collegial and grouped. Each processing unit is associated with its private pool and shares a single global pool with others. Multiple pool algorithms are intended to tackle the bottleneck problem raised by single pool algorithms.

1.4.2 Communication model

In parallel B&B, processing units (PUs) do not work separately, but instead they communicate with each other in order to exchange information during execution. There are three phases requiring communication between PUs in parallel B&B: load balancing, knowledge sharing and termination. In load balancing, a PU which runs out of work needs to communicate with others in order to request works to process. In knowledge sharing, whenever a PU finds a better solution, it broadcasts the new solution to other PUs in order to update the new solution for all of them. This ensures that there is no unnecessary exploration of B&B in other PUs. In termination, all PUs exchange their working status in order to distributively detect whether or not there is no work existing anywhere in the system. The communication can be distinguished between *synchronous* and *asynchronous* algorithms. In *synchronous* algorithms, a computation is split into several iterations. Within an iteration, all PUs process their own B&B subproblems independently. The communication only happens between iterations. In other words, PUs only communicate with each other before moving to the next iteration. In contrast, *asynchronous* algorithms do not define any boundary and the communication can happen at any time during execution.

Besides, these communication models can be combined with the work pool management when designing a parallel B&B algorithm. In our thesis, we only focus on multiple work pool associated with asynchronous communication model as this was shown to be the most suitable for high performance and large scale heterogeneous environments.

1.5 Computing Environments

Before discussing further the main challenges of parallel B&B, we first recall the computing environments considered in this thesis in order to get a complete picture. In practice, computing resources may nowadays be available in the form of cloud, grid, aggregated clusters and personal computers scattered over possibly large scale distributed platforms connected via a network. This huge amount of computational resources offers an impressive computing power which is in theory sufficient to tackle many difficult and large instances of B&B. However exploiting all this impressive power when parallelizing B&B on such platforms is very challenging due to the complexity coming from the system architectures. In our thesis, we consider to use computing resources coming from both shared and distributed memory systems. In the next subsections, we will describe their characteristics and challenges for high performance computing.

1.5.1 Shared Memory Systems

A shared memory system refers to a computing environment where all threads (processors) share the same common memory space. The most common shared memory

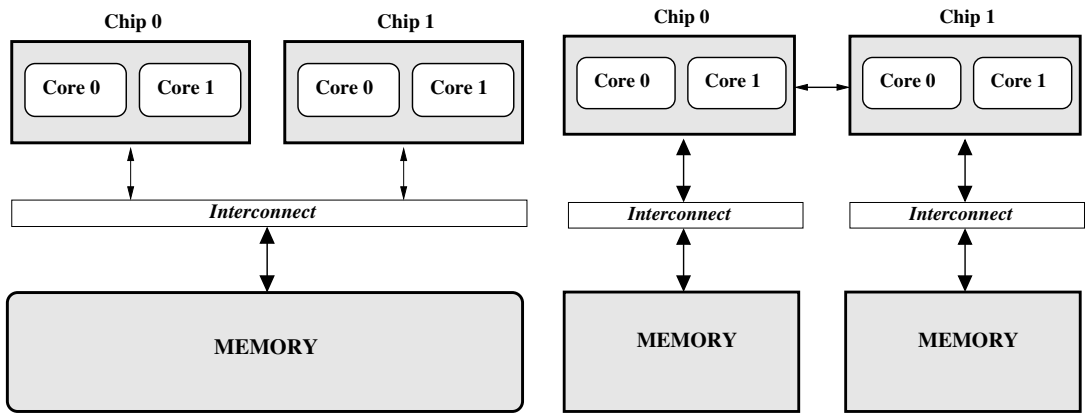

Figure 1.1: Shared Memory System. **Left**: UMA Architecture. **Right**: NUMA Architecture

systems use one or more multicore processors in which a multicore processor has multiple CPUs or cores on a single chip. This architecture assumes that all asynchronous processors are able to access directly any part of main memory thanks to its single logical address space. The interconnection of processors to the main memory defines two types of architectures in shared memory systems. The first one is called uniform memory access (UMA) system, in which all processors connect directly to the main memory. All processors have the same access time to the main memory. This makes UMA architecture easier to program due to the uniform access time to memory. However it also causes scalability issues when many processors are to be handled. Fig 1.1 Left describes a simplified overview of UMA system.

In contrast to UMA system, in a NUMA (non-uniform memory access) architecture, each processor has its own memory block, but each block of memory share the same single address space. It means the same physical address on two processors indicates the same location in memory. Therefore, a processor can have faster access to the directly connected memory. This allows NUMA architectures to scale efficiently because each processor has its own memory controller. Fig 1.1 Right describes a simplified architecture of a typical NUMA system. Nowadays, most of shared memory systems are built based on the NUMA architecture.

Generally, writing parallel programs for either UMA or NUMA architecture requires to coordinate the work of the cores through the shared memory space. This can involve communication among the cores. In fact, all the cores communicate with each other via some local operations on the shared variables. For instance, in parallel B&B, an idle thread running on a core can perform load balancing to migrate some subproblems from another working thread. The idle thread has to remove the subproblems out of the work pool of the working thread and to write them into its own pool. As the number of threads gets higher, the probability that two or more threads access to the same pool at the same time gets also higher.

This can cause data race issues. In order to handle this problem, synchronization techniques are borrowed to transform simultaneous accesses to a sequence of several single access. Although the synchronization allows to protect the shared variables from data race problems, it comes with a price and reduces the potential parallelism of a system. Therefore, overusing the synchronization significantly downgrades the potential parallelism as well as the performance.

1.5.2 Distributed Memory Systems

A distributed memory system indicates that the underlying PUs do not share a common memory but are connected through some kind of networks. The message passing paradigm is the only mean for communication in such system. Generally, the message passing communication is much expensive than the local operations on shared memory systems.

In this thesis, we are interested in large scale distributed systems where PUs are geographically distributed and connected through physical networks (i.e Local area networks or Wide area networks). In fact, this kind of system contains many types of PUs connected by different types of networks. The properties of such systems open several challenging issues for large scale distributed applications. Heterogeneity is the most important issue and it is the main focus of this thesis. Generally speaking, there are two types of heterogeneity [Beaumont 2010a]: node-heterogeneity and link-heterogeneity.

- **Node-heterogeneity**: refers to systems that use more than one kind of processor. In node-heterogeneous computing system, several PUs have different compute powers defined by their hardware architectures. For example, a node can be simply a single CPU, multi-core CPU or a complex one with a multi-core CPU equipped with many GPUs of different potential computing capabilities.

- **Link-heterogeneity**: refers to systems that use different underlying physical networks connecting the underlying PUs. The PUs can thus communicate with different network speeds and bandwidths. There are currently many types of networks such as: ethernet, infiniband, LANs, WANs. etc.

1.5.2.1 Centralized computing architectures

A centralized architecture is widely used when setting up parallel and distributed computing systems and algorithms. In this model, PUs are split into Master and Workers. The master is the central point of the system which usually manages all the control operations. The workers are responsible for most of computation. For example, in parallel B&B, the master usually controls the load balancing to assign subproblems to workers, broadcasts new found solution for knowledge sharing or decides whether to terminate the distributed computation at all workers. The workers perform B&B computation like branching, bounding, selecting and pruning.

We can also find this model in many distributed software such as Napster [Napster], Bittorrent [Bittorrent], Hadoop [Apache]. In fact, this architecture is easy to design and implement, but it was shown to have several shortcomings to the scalability. The centralized master is the single point subjecting to downgrade the performance of the whole system. Since in large scale systems, the master has to process a large amount of requests coming from the workers. If the master does not respond effectively, the workers can waste their computing time by waiting for the responses. For instance, in parallel B&B, whenever a worker is idle, it sends a request to the master and waits for new B&B subproblems. In our experience, we observed that this model is very efficient in small scales as the master is fast enough to handle all the requests of the workers. However, in large scale systems, the performance drops dramatically because of the bottleneck at the master level.

1.5.2.2 Decentralized computing architectures

This model attempts to leverage the bottleneck issue of the centralized architecture as it is fully distributed and does not count on any centralized master. There is no difference between master or worker in this model. Each PU can play the role of both master and worker so that the communications are evenly distributed among them. This model can be classified according to logical overlay or topology induced by PUs communications:

- **Unstructured Overlay**: there is no specific overlay topology imposed globally upon all PUs and the topology is often random. Each PU broadcasts messages to its neighbors when sending a request. The broadcast is repeated until the sender receives the answer or a maximum number of flooding is reached.

- **Structured Overlay**: an overlay topology is used to structure the PUs (or peers) in the system. The overlay structure aims at routing traffic effectively through the network. PUs in a structured overlay have to maintain a list of neighbors that satisfy a given criterion. Stoica *et al.* [Stoica 2001] firstly proposed to organize the PUs as a Ring and use Distributed Hash Table (DHT). Later on, the DHT is adapted in many works (e.g YaCy [Yacy], CoralCDN [CoralCDN], Kademlia [Maymounkov 2002].

In this thesis, we also studied the impact of a structured overlay on the performance of parallel B&B in distributed large scale systems. In particular, we organize all available PUs as a tree and use the tree overlay structure to handle all the communications of parallel B&B such as load balancing, knowledge sharing and distributed termination detection in a minimal cost.

1.6 Parallel B&B for Distributed Computing Environments

In the previous sections, we briefly presented some characteristics of the considered computing environments, sequential/parallel B&B as well as some general implementation issues. In this section, we will firstly detail specific challenges of parallel B&B algorithms, then we will present some related works of parallel B&B on both shared and distributed memory systems.

1.6.1 Main challenges in parallel distributed B&B

1.6.1.1 Irregularity

At a first glance, it is straightforward to parallelize a B&B algorithm because each of the generated subproblems can be solved independently. However, the tree-based parallelization yields highly irregular computations and raises difficult challenges:

- tasks are created dynamically in the course of the algorithm.
- tasks are assigned to PUs at runtime.
- the structure of the B&B tree to explore is not known beforehand.
- workload of PUs varies dynamically during execution.

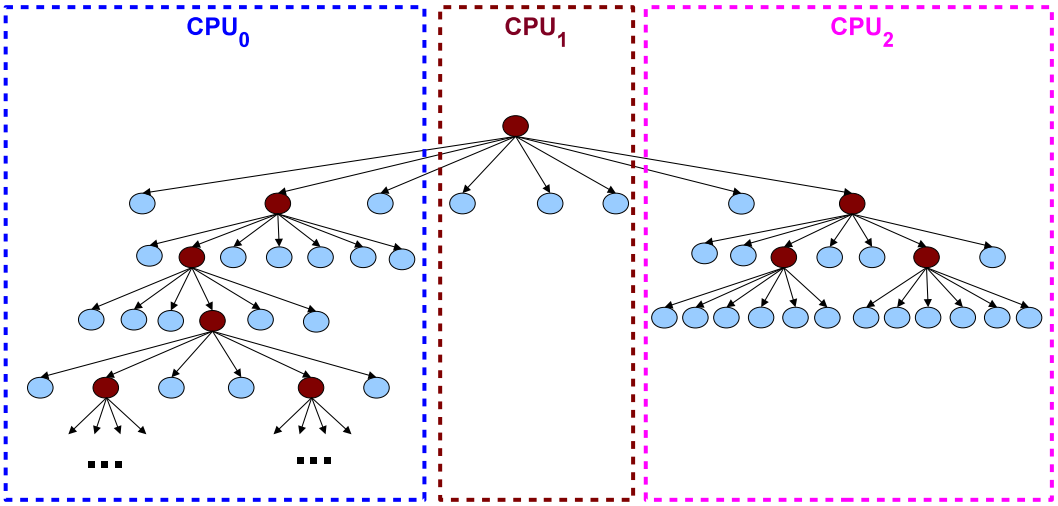

Figure 1.2: An Example of Parallel B&B on 3 CPUs

A B&B process can be simply viewed as a search tree exploration where the original optimization problem is the root node of the tree. The root node is processed to generate several children nodes. After being processed, some of the children nodes continue to generate new children, and the others are completely explored and do

not create any children due to the pruning operation of B&B. During the execution, many new nodes are recursively created and this makes the corresponding tree grow larger and larger. In parallel B&B, the tree-based computation is handled simultaneously by several PUs. For instance, Fig 1.2 illustrates a simple example of a parallel B&B computation on 3 CPUs. Firstly, CPU_1 processes the root node (i.e the original optimization problem) and generates several children nodes. CPU_0 and CPU_2 perform a load balancing operation to fetch some nodes from CPU_1. Although at this time all the CPUs have the same workload and process approximately the same number of nodes, CPU_0 is eventually the most overloaded and the others are underloaded during the computation. In fact, CPU_0 performs the B&B computation for a very large part of the B&B tree, but CPU_1 and CPU_2 only handle a relatively small part. This problem comes from the irregular nature of parallel B&B as the tree structure is unknown in advance and it is shaped at runtime. Therefore, several processing units are under-utilized and few processing units are overloaded to process works during the computation of parallel B&B. The workload irregularity of B&B can significantly downgrade performance and prevent high performance and scalability which makes it challenging to design efficient parallel B&B.

1.6.1.2 Knowledge sharing

Following the workload irregularity issue, designing a scheme for exchanging the best solution found so far among processing units is another major issue in parallel B&B algorithms. In fact, it is crucial to share any new best solution discovered by any processing units at runtime in order to quickly prune unnecessary branches which could not lead to the optimal solution. The strategies for updating the new best solution depend strongly upon the architecture being used. For instance, in shared memory systems, a global variable, which is accessible by all threads, is used to store this global information. When a thread finds a better value, the thread will update to the global variable that makes all other threads be aware of the new value. In distributed memory systems, message passing is the only mean to use for communication among processing units. Therefore, whenever a processing unit finds a better value, it must send messages to others in order to update the new found value into their local variable. In both cases, a strategy has to ensure that the global information is broadcasted to all other processes and the performance of such a broadcast is clearly dependent on the interconnect of a given architecture. In this thesis, we use the centralized approach for shared memory systems and a distributed approach based on a tree overlay for distributed memory systems in order to broadcast the new best found solution so far to all PUs in the system.

1.6.1.3 Termination detection

Termination detection is one central issue in parallel B&B. It consists in determining the state when all works are completed at all processing units (PUs). Similar to the knowledge sharing, the strategies for detecting termination depend on the underlying computing environments. In general, termination algorithms must satisfy the

following aspects: (i) correctly detect whether or not some work still exists somewhere in the system, (ii) run efficiently and lightly with minimal communication cost to not disturb useful computations. In shared memory systems, a centralized approach can be employed. A global barrier allows threads to check-in and check-out in order to nominate the termination. When a thread runs out of works, it checks-in to the barrier. If a thread notices that it is the last one to check-in, it will inform the termination to other threads. When a thread fetches works from others, it checks-out from the barrier. In distributed memory systems, as the centralized approach is not scalable as the number of PUs increases, a distributed scheme is usually employed. In this thesis, to detect termination distributively, we employ a distributed scheme based on a tree topology.

1.6.2 B&B on shared memory systems

Barreto *et al.* [Barreto 2010] conducted a comparison of parallel B&B approaches on shared and distributed memory systems as well as their potential improvements compared to the sequential one. The authors used OpenMP and MPI to implement their approaches on shared and distributed memory respectively. In the OpenMP approach, a single work pool storing all tasks (i.e generated subproblems of B&B) is shared by all worker threads. In order to avoid the data race situation at the pool, a synchronization technique is chosen. It properly handles multiple accesses to the single pool either when idle threads try to retrieve tasks or when working threads push new tasks. In [Barreto 2010], the authors reported a good speedup of OpenMP and MPI approaches compared with the sequential one. However, the speedup of MPI was found to be slightly better than the OpenMP approach which may appear counter-intuitively at the first sight. It is in fact well understood that inter-communication is much costly in MPI since the messages have to pass through standard networks connecting computers. However, this result also enlightens the negative impact of using synchronization mechanisms in shared memory systems. It can be explained as following. Firstly, high parallelism can not be achieved since all working threads are serialized while accessing the single pool. Secondly, the overuse of synchronization also introduces an important overhead.

Casado *et al.* [Casado 2008] proposed two schemes for parallelizing B&B algorithms on shared memory multicore systems. In the first scheme, so-called Global-PAMICO, all threads share a global pool of generated subproblems therefore a synchronization mechanism is used to synchronize the accesses of all threads to the global pool. In the second scheme, so-called Local-PAMICO, each thread manages a local pool to avoid a significant overhead of the synchronization caused by the global pool of the first scheme. Besides, a thread terminates itself when there is no more subproblems in its local pool. A dynamic load balancing is proposed to deal with the irregularity of B&B as following. At each iteration, when a certain condition is satisfied, a thread will generate a new one and migrates work from its local pool to the pool of the new thread. The condition is described as following: the number of running threads are less than the total number of available cores and

there is more than one subproblem in the local pool of the thread. This strategy might also introduce a significant overhead when creating a lot of new threads due to the irregularity of parallel B&B.

Similarly, in [Mezmaz 2013], the authors presented an approach to parallelize B&B on multicore systems. In this approach, all threads also share a single work pool. They reported a big gap between their implementation and the ideal linear speed up. In fact, the larger the number of threads/cores are used, the bigger the gap is. There are many reasons behind the scene however synchronization is the biggest issue.

In [Evtushenko 2009], the B&B solver (BNB-Solver), a software platform allowing the use of serial, shared memory and distributed memory B&B algorithm is presented. BNB-Solver is based on the parallel exploration of the tree and assumes that several CPU threads are used where each thread has its local pool of subproblems and shares a global pool of subproblems with other threads. In BNB-Solver, each thread executes a fixed number of N iterations of the sequential B&B algorithm. During the N iterations, each thread stores the generated new subproblems in its local pool. It only transfers a part of subproblems from the local pool to the global pool at the end of N iterations. Each thread tries to select from its local pool the next subproblem to be processed. If the local pool is empty, the thread selects a subproblem from the global pool. If the global pool is empty, the thread blocks itself until another thread puts at least one subproblem in the global pool. Once new subproblems are inserted into the global pool, blocked threads are released and subproblems are retrieved from the global pool. BNB-Solver ends when the global pool is empty and all threads are blocked.

Savadi *et al.* [Savadi 2012] introduced an approach taking into account the memory hierarchy of multicore systems. The authors proposed to map task tree of B&B to the memory hierarchy tree of multicore systems. In the memory hierarchy tree, non-leaf nodes and leaf nodes are mapped on cache/memory components and the processor cores respectively. A node of the task tree is mapped on a level of memory tree which has equal or larger memory size than needed memory for the node execution. Furthermore, the mapping should respect the locality of the node and its children so that it tries to reduce execs communication between levels of memory tree.

Recently, Silva *et al.* [Silva 2014] proposed a model called Multicore Cluster Model which captures some relevant performance characteristics in multicore systems such as the influence of memory hierarchy and contention. In the model, three communication models are taken into account: i) the communication made through shared memory by intra-chip cache, ii) through inter-chip shared memory and iii) communication between cluster nodes via messages. Furthermore, a load balance is also proposed to schedule tasks based on the available amount of the cache memory so that it can avoid memory contention which negatively affect performance. An improvement is reported when the model is used to implement a parallel B&B algorithm for the Set Partition Problem.

1.6.3 B&B on distributed memory systems

Parallel B&B on distributed memory systems have also attracted much attention from the community. In this context, three paradigms are widely used: Master-Worker B&B, Hierarchical Master-Work B&B and Decentralized B&B. A non exhaustive set of representive approaches are described in this section.

1.6.3.1 Master-Worker B&B

The Master-Worker (MW) paradigm is the most popular technique for designing distributed applications. This paradigm appears in many applications in both academic and industry since it is simple in terms of design, implementation but produces quite good performance. SETI [Korpela 2001] is one of the first large scale distributed computing system which is based on the MW paradigm. SETI employs the power of volunteer computing, which comprises of large number of home computers connected by internet, to analyze scientific data from radio telescope. The data are parsed in small chunks that are sent to home computers (i.e workers) for off-site computation. Then the results will be sent back to the SETI facility (i.e. master). This paradigm is also being widely used to solve large and difficult B&B problems. For instance, some frameworks [Chen , Goux 2000] allow users to run parallel B&B computation on the grid.

Mezmaz *et al.* [Mezmaz 2007b, Mezmaz 2007c] proposed B&B@Grid for large scale B&B algorithm using the MW paradigm. The authors proposed a compact encoding for subproblems of B&B. A list of subproblems is encoded by *fold* operators into a unique interval of two integers. On the other hand, an interval is decoded by a *unfold* operator to a list of corresponding subproblems. This approach tends to reduce the size of data transferred through the networks in the grid while managing load balancing among master and workers. Though interval representation brings many advantages, there are few disadvantages. Firstly, the fold/unfold operations introduce a non-negligible overhead. Secondly, the interval representation is not suitable for workers with SIMD architecture (i.e. GPUs).

Otten *et al.* [Otten] introduced a scheme to predict the complexity of a subproblem so that subproblems can be evenly distributed among workers based on the prediction scheme. Initially, the master node explores the tree up to a parallelization frontier for having enough subproblems, then send them to its workers and wait for the result. Load balancing is ensured by their prediction scheme which estimates the size of the explored space of a subproblem (i.e. the number of explored nodes to solve a subproblem) so that load is evenly distributed among workers.

Recently, [Budiu 2011] introduced a DryadOpt library which enables massively parallel and distributed execution of optimization algorithms on distributed data-parallel execution engines (DDPEE) such as Dryad [Isard 2007], MapReduce [Dean 2008] or Hadoop [Apache]. In fact, DDPEEs bring high parallelism but they impose some important restrictions on the algorithms which can be implemented. Algorithms suited to DDPEEs need to process a large amount of independent data and processes run in different memory address spaces can not communicate until completed.

Communication and data exchanges can only happen when a round of computation among processes terminate. The DryadOpt adapted the synchronous model, presented in previous section, in order to solve optimization problems on DDPEEs as following. Generally, the parallel B&B based on DryadOpt can be interpreted as a sequence of computations, which is called round, and communication for load balancing only occurs between two consecutive rounds. Firstly, the client workstation plays the role of the master and reads a problem instance and run a sequential B&B solver to generate a large frontier containing enough tasks for cluster machines. The tasks in the frontier are partitioned in disjoint sets and each set will be processed by each machine independently. Then each round, which takes the previous frontier as an input, is executed on cluster machines to produce a new frontier. After a round of computation, control is returned to the client workstation which decides either to start a new round or to terminate. The computation is terminated whenever the frontier is empty. Moreover, load balancing is ensured at the end of each round by repartitioning the tasks in the frontier such that each machine processes approximately the same amount of work. This solution can be applied on machine level as well as core level. Even though the DryadOpt brings a promising result, it does not take into account the computational grid environments.

[Kouki 2010] proposed an algorithm called GAUUB which is based on the master-worker paradigm that distributes tasks to all worker processors on grid computing environments. The lower bound algorithms [Ladhari 2005] is firstly used by the algorithm GAUUB. Moreover, the authors reported that the instances Ta42 and Ta50 were firstly solved by 50 processors. Then the authors improved the algorithm GAUUB by proposing another version called GALB [Kouki 2012, Kouki 2013] that reduces load unbalancing among the available processors. The algorithm GALB is composed of two steps: initialization and distribution. The master executes the initialization step by performing the sequential B&B algorithm until reaching a fixed level L of the search tree in order to generate a large amount of work to distribute among the slaves processors and therefore to ensure load balancing for all processors. All slaves execute the distribution step by exploring the assigned tasks locally and update new found better solution to the master.

1.6.3.2 Hierarchical Master-Worker B&B

Generally, the performance of a MW-based system strongly depends on the capability of the master. If the master can handle efficiently requests coming from workers, then the system can bring very good performance, otherwise the performance will be significantly decreased. Therefore, it is well understood that MW-based approaches are only suitable at small or intermediate scales. Some hierarchical master-worker (HMW) schemes are proposed to solve the scalability issue of MW. Basically, despite many variations among the HMW approaches, their key idea is to construct several masters in a hierarchical manner (like a tree) in order to leverage communication bottleneck on a single master. In HMW paradigm, there are two layers: a control layer comprising of one or more levels of masters and a work layer composing

workers.

Aida *et al.* [Aida 2002, Aida 2005] proposed a three-tier tree structure comprising a supervisor, masters and workers. The supervisor is the root node of the tree and manages all masters. Workers are grouped into sets and each set is managed by a master. The supervisor and the masters are in charge of all communications among workers belonging to different sets such as load balancing, broadcasting upper bound values. The communications are performed along the upward and downward paths of the tree. The supervisor splits the B&B tree into several disjoint parts and assigns each of them to a master. The master of each set then assigns work units to its workers for computation. The workers execute the B&B computation and request more work units from the master when they run out of work. When all the workers completely process the assigned B&B tree part, their master will then ask the supervisor for another part. The authors discussed the granularity of tasks, especially when tasks are fine-grained, the communication overhead is too high compared to the computation of tasks.

Xu *et al.* [Xu 2005] proposed a framework to implement parallel search algorithms called APLS. The framework uses: Master, Hub and Workers. A hub controls a fixed number of workers and the number of hubs increases proportionally with the number of workers. Load balancing is also taken into account at two levels: intra-cluster and inter-cluster. In the first level, the hub manages dynamic load balancing. Workers periodically update their workload to the hub, hence the hub can try to balance workload among workers when it detects an unbalanced situation. Besides, workers also sends work requests to the Hub when workload is lower than a threshold. Upon receiving the message, the hub asks the most overloaded worker to share some works (i.e subproblems) to the requesting one. Similarly, the master is responsible to balance workload among hubs. This design principle can also be found in PICO [Eckstein 2001] or in [Drozdowski 2012].

Diconstanzo [Diconstanzo 2007] described a HMW approach composed of four types of components: master, sub-master, worker and leader. The master, sub-master, and worker components have similar functionality as the other HMW approaches presented above. The main difference is the leader components. Each set of processes is deployed on a physical cluster, and a leader is chosen among workers. The role of the leader is to handle communication of broadcasting the upper bounds among groups of workers. When a worker finds an upper bound, it broadcast the new solution to all workers of the group, and only the leader is in charge of forwarding the new solution to other leaders of other groups in order to broadcast to all workers in all groups.

More recently, in [Bendjoudi 2012b, Bendjoudi 2012c], the authors suggest a MHW architecture which allows workers to communicate directly together after receiving a task from the master. The proposed approach tends to avoid up/down communication among workers of different group while performing load balancing in MHW so that tasks are distributed more quickly from one group to others compared with the standard communication manner in MHW. Furthermore, this mechanism also exposes less communication bottleneck to masters and implicitly employs a

peer-to-peer communication among workers.

1.6.3.3 Decentralized B&B

The MW paradigm only works well at small/intermediate scales and often faces the scalability issues due to communication bottlenecks around the master. The performance of MW systems strongly depends on the capability of the master. To overcome this limitation, the HMW paradigm considers to use multiple levels of masters to handle the whole system so that the communication load can be distorted evenly among the masters. However, these approaches do not solve completely the scalability issues as detailed in the following. Firstly, scalability of a HWM system strongly depends on the proportion between master and workers. If this ratio is not well chosen, the communication bottleneck can occur at sub-masters and downgrade the overall performance. Secondly, the control layer of a HWM system are mainly responsible for communication, hence all potential computational power of this layer are wasted. Lastly, all communication have to pass through the masters of the control layer

On the other hand, a fully distributed peer-to-peer paradigm overcomes this limitation by distributing communication load among all processes in the system. In distributed B&B, the asynchronous multiple pool paradigm is considered as a good design and implementation choice. Each peer has a local pool storing all work units for processing. If the local pool of a peer is empty, the peer will send work requests to others according to the corresponding load balancing algorithms. Upon receiving a work request, a peer will share some work units from its local pool to the requesting peer.

In [Luling 1992, Tschoke 1995], the load balancing is performed on each processor in parallel to the sequential B&B algorithm. The authors proposed a load balancing algorithm based on workload. The basic idea is to balance the workload of a peer with its neighbors. A peer and its four neighbors collaborate to form an island. The peers of an island help to balance workload each other whenever some of them are under/over load (i.e workload difference among them reach a predefined situation). Besides every peer belongs to several overlapped islands, the workload balance through the whole system is therefore achieved. Furthermore, in this approach, the workload is defined by a weight function which takes both quality and quantity of generated subproblems into account. The quantity is simply the number of subproblems in a local pool. The quality is interpreted as the minimum cost of generated subproblems of a local pool since they want to keep all best subproblems (with current best lower bounds) in every local pool in order to provide best-first search and avoid search overhead as much as possible.

Similarly, some fully distributed B&B were presented in [Finkel 1987, Iamnitchi 2000, Diconstanzo 2007, Djamai 2011a]. Iamnitchi *et al.* [Iamnitchi 2000] proposed to piggyback the best solutions to the most used messages in order to efficiently broadcast the solution over the whole system. Di Constanzo *et al.* [Diconstanzo 2007] proposed to define a peer-to-peer infrastructure at the communication layer for routing

messages in a HMW system. A message is routed through multiple intermediate peers to reach the final destination. The objective of this approach is to reduce communication bottleneck at the masters, but it introduced unnecessary additional delays for sending messages. Therefore, the system can not be utilized at its full potential capacity. Djamai *et al.* [Djamai 2011a] proposed an approach for very large scale systems (up to hundred of thousand peers) and provided new work sharing mechanism, termination detection. Beside the authors also provided a proof of correctness of their approach.

1.7 Conclusions and discussions

Solving large instances COPs is time-intensive because of their large search space. The B&B algorithm appears to be a good solution in this context as it aims to efficiently reduce the search space by eliminating unnecessary regions. Despite the search space is significantly decreased, it is still too large to be completed in a reasonable time by the sequential B&B. Consequently, sequential approaches often fail providing good performances. Within this context, parallel and distributed B&B are a key solution to tackle the underlying compute challenge. B&B computations can be distributed to many threads of the same compute node, or many compute nodes of the same cluster, or many clusters of the same grid, or even many compute nodes geographically distributed and connected via internet. On one side, the distributed resources coming from cluster, grid or cloud provide an impressive aggregated computing power. On the other side, parallel B&B exposes several sources of parallelism ranging from low to high level, from node-based to tree-based parallelism which allow parallel B&B to be effectively implemented on different types of computing resources associated with different architectures. In theory, B&B exposes several sources of parallelism. In practice, there exists several challenging issues which might significantly downgrade the performance of parallel B&B when executing on complex distributed systems.

- Workload Irregularity: parallel B&B workload is too irregular to be predicted beforehand. B&B work is generated at runtime which can push most of the workload on few processing units while the others are starving. This issue is the major cause of poor performance and low scalability. Let us remind that good performance and high scalability are only achieved when the workload is evenly distributed during the execution. Dynamic load balancing approaches seem to be a good answer to this problem as they try to balance workload of computing systems at runtime. Previous works aimed to distribute workload on several computing platforms, however dynamic load balancing was not *the* main concern.

- Heterogeneity: complex computing systems harness different computing resources with different architectures interconnected by different networks. Node-heterogeneity and link-heterogeneity, which are rarely addressed in a comprehensive manner, also have a strong impact on the performance of parallel B&B.

We also take into consideration these heterogeneity aspects when designing a dynamic load balancing algorithm.

In this thesis we mainly focus on tackling the workload irregularity of parallel B&B as well as the heterogeneity of the distributed underlying computing systems while solving difficult and challenging COPs. In the following chapters, we will study the impact of different dynamic load balancing schemes for parallel B&B on different distributed computing systems ranging from homogeneous to node-heterogeneous and link-heterogeneous systems. We then discuss our solution for each scenario.

The remainder of this thesis is organized as the following. Chapter 2 describes in detail our solutions for dynamic load balancing while running parallel B&B on homogeneous systems. Chapter 3 and 4 describes our proposal for balancing irregular workload of parallel B&B on node-heterogeneous and link-heterogeneous systems.

Chapter 2

Dynamic Load Balancing for Homogeneous Computing Environments

Contents

Main publication related to this chapter

T. Vu, B. Derbel, A. Ali, A. Bendjoudi and N. Melab. Overlay-centric Load Balancing: Applications to UTS and B&B. *14th IEEE International Conference on Cluster Computing, CLUSTER 2012*, page 382-390, Proceedings, IEEE, 2012.

2.1 Introduction

Most existing recent works on parallel B&B in distributed computational environments are based on Master-Worker [Mezmaz 2007a], Hierarchical Master-Worker [Bendjoudi 2012a] and Peer-to-Peer [Djamai 2013] architectures. Let us recall some key contributions of these works in this context. In [Mezmaz 2007a], the authors proposed an efficient mechanism to transform a B&B tree to a compact representation and *vice versa*. This mechanism reduces communication cost among processing units. The authors implemented their approach using MW scheme. In [Bendjoudi 2012a], the authors focus on the bottleneck issues of the MW architecture by adding some intermediate layers of masters to control all workers in the system. They handle communication among workers of different groups by enabling direct communication among them. Similarly, [Djamai 2013] proposed a fully distributed protocol for B&B in order to overcome the scalability issues faced in the centralized approaches. However, the authors mainly focused on the correctness of distributed termination of their distributed protocol. Despite several important issues of parallel B&B tackled in the recent works, the irregularity issue still lacks attention in the literature. We argue that this is the most severe issue of parallel B&B leading to poor performance in distributed and massively parallel systems.

Our main objective is to tackle the irregularity issues and to ensure a good load balancing in computing systems during execution time. Good load balancing refers to good utilization of computing units and results in good speedup of the parallel application. Achieving a good load balance in the presence of irregularity in parallel applications is a very challenging task. There are many issues that need to be taken into account when choosing an accurate load balancing method, especially in large scale computing systems.

The focus of this chapter is to present, design and implement dynamic load balancing protocols to solve parallel B&B on homogeneous distributed computing systems. The main objective is to solve the irregularity of the B&B application effectively and achieve high performance in large scale systems. The remainder of this chapter is structured as follows. Section 2.2 presents some classifications of dynamic load balancing algorithms. We further analyze and highlight that work stealing is a good candidate in the context of this thesis. Section 2.3 describes the design and the implementation of random work stealing for parallel B&B in distributed settings. Section 2.4 details about our approach. Section 2.5 briefly introduces two application benchmarks (i.e the Taillard Flow-Shop scheduling problems and the Unbalanced Tree Search problems) which are used not only in this chapter, but also

in the remainder of this thesis. Finally, Section 2.6 presents our experimental results and Section 2.7 concludes the chapter.

2.2 Dynamic Load Balancing

2.2.1 Overview

Balancing workload among processing units (PUs) is one of the most important challenges when developing parallel irregular applications, especially in distributed computing systems. In the context of parallel B&B, balancing workload refers to offload subproblems from overloaded PUs to underloaded/idle PUs in order to ensure that all PUs have approximately the same workload. If the subproblems are not evenly distributed among PUs, then some PUs may run out of work and become idle while others have large number of subproblems to process. This issue is referred to as workload unbalance, and it is believed to majorly downgrade the overall performance of computing systems if not handled efficiently. Furthermore, this problem is more pronounced with the irregularity of parallel B&B where subproblems are created dynamically during execution. Consequently, good speedups and parallel efficiency can not be directly achieved in a straightforward manner.

Dynamic load balancing methods are well suited for tackling workload unbalance occurring at runtime. These methods [Burton 1981, Eager 1986, Kumar 1994, Shivaratri 1992a] are fully distributed and often implemented in asynchronous communication with multiple pools. In more details, each PU is associated with a single pool for storing generated subproblems and communicate with others at runtime for load balancing. Having this in mind, dynamic load balancing methods can be distinguished in three types, depending on who is the initiator:

- **Sender-initiated** or **work pushing** [Eager 1986]: refers to the class of methods where overloaded PUs are in charge of balancing workload. In these methods, a sender PU automatically offloads tasks to an appropriately chosen PU. The task offload occurs when the sender PU has more tasks than a threshold value in its work pool. These techniques aim at minimizing idle time of PUs because tasks are pushed ahead before they are actually needed but at the cost of additional communication overheads. Before offloading the tasks, the senders have to answer these questions: How to select a target PU? How many tasks to offload? How often to offload tasks?

- **Receiver-initiated** or **work stealing** [Burton 1981]: refers to the class of methods where idle PUs perform load balancing operations. In contrast with work pushing methods, tasks are not automatically offloaded ahead between PUs. When a PU finds no task in its local pool, it sends work requests to an appropriately chosen PU for effectively retrieving tasks. Upon receiving a request, a PU will offload tasks if its own work pool is not empty. Otherwise, the receiver PU rejects the work request and the sender PU repeats the process until fetching some tasks. This approach aims at minimizing communication

overheads at the expense of idle time of PUs. Similarly, some questions have to be answered: How to chose an appropriate target PU? How many tasks to offload?

- **Hybrid** [Shivaratri 1992a]: combines the previous two approaches into a comprehensive one. Tasks are offloaded to PUs needing them, or on demand when PUs are idle.

The policy of choosing appropriate PUs in these approaches is the most important factor for good load balancing. Besides, there are some variants of dynamic load balancing algorithms according to the target selection policy. In work pushing, three variants have been proposed in [Eager 1986]:

- *Random*: a sender PU offloads a task to another PU selected at random whenever the local pool of the sender exceeds a predefined threshold value.
- *Threshold*: a sender PU offloads a task to another PU chosen at random if it finds that the local pool of the selected one smaller than a predefined threshold value.
- *Shortest*: a sender PU randomly picks a set of PUs and offloads a task to the one owning the shortest local pool.

Although, work pushing produces good performance, there are sill some drawbacks in the algorithm. Good load balancing strongly depends on how often tasks are offloaded in a system. Work pushing is controlled through the use of a predefined threshold. If this threshold is not well optimized to the correct value, tasks will be moved ahead so often or so rarely. For instance, if the value is larger than needed, very few offload operations are performed. If the value is smaller than needed, many task are offloaded in advance causing communication overhead and waste of bandwidth. Furthermore, work pushing is unstable when solving large instances. Although the system load is high and all PUs are kept busy, task offloading still occurs. Therefore, work pushing is not a good choice for irregular applications like parallel B&B.

[Shivaratri 1992b] proposed the hybrid approach which improves both work pushing and work stealing. In this approach, a PU uses current load information of others to decide where to look for tasks and where to offload tasks. However this approach is not suitable because the load information changes very quickly over time in irregular applications like parallel B&B.

Work stealing techniques seem to be more suitable for irregular applications and give a better overall load balance. In the work stealing approach, communication is only enabled when PUs are idle. Work stealing is stable because there is no communication as the system load is high. Therefore, it is chosen as a scheduler for load balancing in many frameworks (e.g Cilk [Frigo 1998], Intel TBB [Intel], OpenMP 3.0 [OpenMP 2011] and Javelin [Neary 2000]). We mainly focus on work stealing in the rest of the thesis.

2.2.2 Work Stealing

Work stealing was firstly proposed by Burton et al. [Burton 1981]. Later on, it appeared in many parallel frameworks for scheduling work load while executing parallel applications, both on shared memory and distributed memory systems. In work stealing, a PU, which runs out of work and has no task in its local pool, is called *thief*. A thief attempts to retrieve tasks by sending a steal request to an appropriately chosen PU. The selected PU is called *victim*. Upon receiving a steal request, a victim PU will service the request by offloading some tasks to the corresponding thief if its local pool is not empty. Otherwise, a reject message is sent back to the thief and the thief repeats steal attempts until fetching some tasks. Figure 2.1 sketches the status of PUs during execution of work stealing.

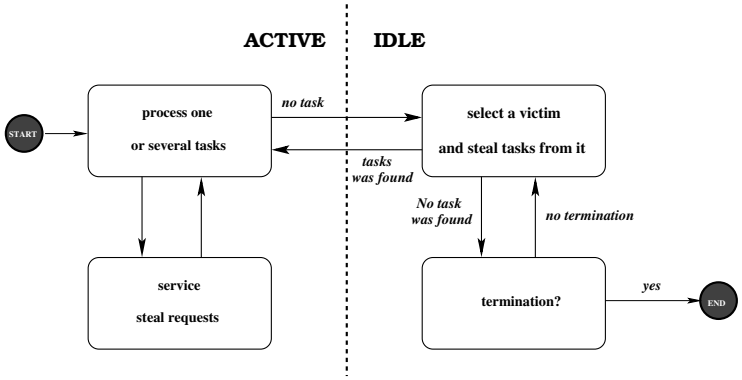

Figure 2.1: State Diagram of Work Stealing

Algorithm 2 briefly describes a base implementation of the algorithm. Let us remark that the victim selection and work sharing policy are the heart of the algorithm. The victim selection policy defines how a thief chooses a victim in order to fetch works as quickly as possible, leading to minimize idle time. The work sharing policy defines how a victim shares tasks with a thief so that the workload will be eventually balanced between them resulting in maximizing useful computations.

In particular, there are some variants of work stealing in homogeneous computational environments based on the victim selection policy:

- *Random*: it is the simplest form of work stealing where a thief chooses a victim uniformly at random. Hence the stealing probability is a constant value and it is the same for any PUs.

- *Local Round Robin*: each PU locally manages a separate variable *nextTarget* pointing to the next victim. When a PU becomes a thief, it sends a steal request to the victim pointed by the local variable *nextTarget*. The variable *nextTarget* is then increased in round robin manner (e.g $(nextTarget+1)$ mod N, where N is the total available number of PUs in the system). Therefore, different PUs probably store different values for the variable *nextTarget*.

- *Global Round Robin*: it is similar to the Local Round Robin scheme, but instead of storing the variable *nextTarget* locally at each PU, the variable nextTarget is globally shared among all PUs. Thus all PUs share the same value for the variable *nextTarget*.

Algorithm 2: Work Stealing Template

```
1 while termination not detected do
2     if local queue is empty then
3         Execute Procedure THIEF;
4     else
5         Execute Procedure VICTIM;
```

Procedure Thief

```
1 repeat
2     u ← VICTIM_SELECTION;
3     SEND a steal request message to u;
4     RECEIVE u's response (reject or work) message ;
5     if a steal request is pending then
6         v ← pull the next pending thief request;
7         Send back a reject message to v ;
8 until retrieving work from victim u or termination detected;
```

Procedure Victim

```
1  if a steal request is pending then
2      if tasks are available then
3          v ← pull the next pending thief request;
4          work ← SHARE_WORK;
5          Send back shared work to v ;
6      else
7          Send back a reject message to v ;
8  else
9      task ← pop the next task from local pool;
10     EXECUTE task ;
```

In theory, Blumofe *et al.* [Blumofe 1999] proved that the *Random* policy is optimal according to some constraints on the communication model. In practice, the random policy produces very good performance in homogeneous computational environments [Dinan 2007, Dinan 2009]. In this thesis, we propose to apply the work stealing paradigm for B&B. Let us notice that despite a rich references of parallel B&B

in the literature [Mezmaz 2007a, Bendjoudi 2012a, Djamai 2013, Chakroun 2013a], the work stealing paradigm has not been considered in order to solve the workload irregularity of parallel B&B while running on distributed computing environments.

2.3 Random Work Stealing and Parallel B&B

2.3.1 Preliminaries

To simplify the presentation let us model the B&B algorithm, as a tree search algorithm starting from the root node which is the representation of an optimization problem. During the search, a parent node generates new child nodes (e.g., representing partial/complete candidate solutions) at runtime. The quality of these nodes is evaluated (bounding) using a given (heuristic) procedure. Then, according to the search state, some nodes are discarded (pruning) whether some others can be selected and the tree is expanded (branching) to push the search forward and so on. Having this in mind, the general architecture of our approach for distributing search computations is depicted in Fig. 2.2

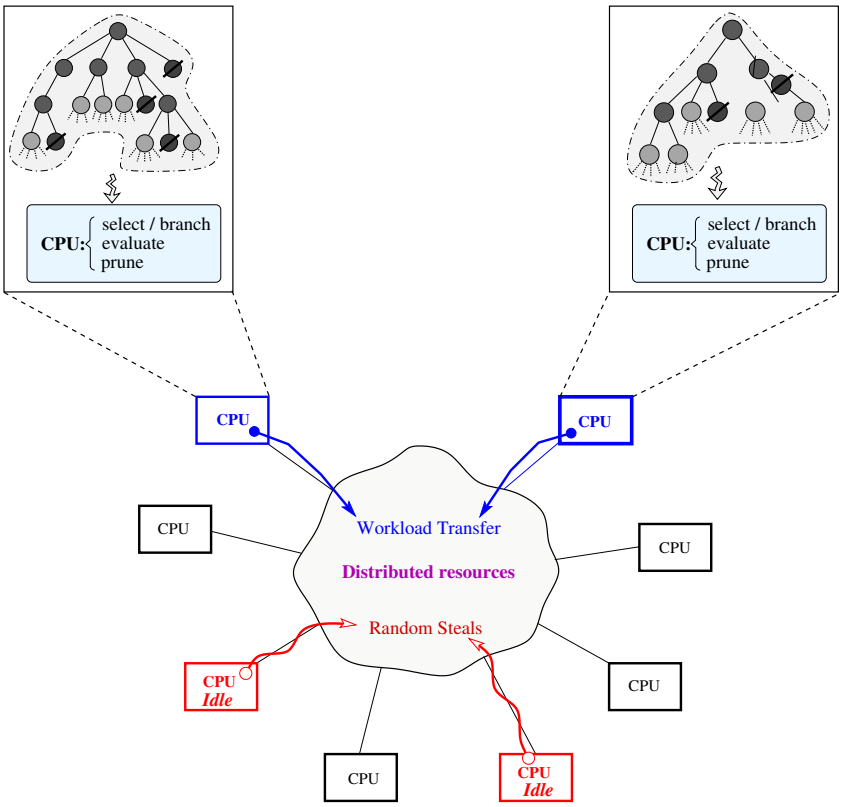

Figure 2.2: Overview of Random Work Stealing for Parallel B&B

Let us remark that there are three challenges in parallel B&B: irregularity, knowledge sharing and termination detection. The irregularity comes from the nature of

the applications as presented above. We aim at tackling this issue by using the random work stealing (RWS) algorithm.

2.3.2 Work sharing

One crucial issue in RWS for efficient dynamic load balancing is the amount of work, denoted f, to be transferred between thieves and victims. Generally, the thief attempts to balance the load evenly between itself and the victim. In fact, when this amount of work is very small, the large overhead is observed since many load balancing operations are performed. At the opposite, when it is very large, too few load balancing operations will occur, thereby resulting in large idle times despite the fact that surplus work could be available. In classical RWS approaches, this is a hand-tuned parameter which depends on the distributed system and the application context [Min 2011]. In a theoretical study [Blumofe 1999], the stability and optimality of RWS can be analytically guaranteed for $f \leq 1/2$. In practice, the so called steal-half strategy ($f = 1/2$) is often shown to perform efficiently using homogenous computing units. Therefore, we decided to use steal-half for balancing workload between thieves and victims in our distributed computing system.

2.3.3 Knowledge sharing and termination detection

Knowledge sharing and termination detection are also the two crucial parameters which may impact the overall performance of parallel B&B. We map all available PUs to a binary tree as presented in Fig 2.3, and use this tree to tackle the two issues effectively. In the following two subsections, we describe in detail how these issues are addressed by the overlay.

2.3.3.1 Knowledge sharing

In distributed B&B, broadcast communication plays an important role for sharing knowledge among PUs in the system. For instance, a PU, which finds a new upper bound, must broadcast it to other PUs in order to update the new upper bound for all PUs. It is a mandatory phase in B&B to avoid unnecessary exploration of branches in other PUs (i.e. problems whose evaluated bound value is worse than the newly found best solution). To perform a broadcast in a simple and efficient manner, we use a hierarchical broadcast mechanism. All PUs are mapped into a *binary tree* overlay. The broadcast is implemented as following:

- If a new upper bound improves previous best known solution is found locally by a PU, then it sends the new upper bound to its parent ($1 - to - 1$ communication) and children ($1 - to - N$ communication).
- If a PU receives a new upper bound from either its children or its parent, the PU will compare the new upper bound with its local best solution. If the new upper bound improves the local best solution, then the PU will update the new value to its local variable and forward the information to its parent and

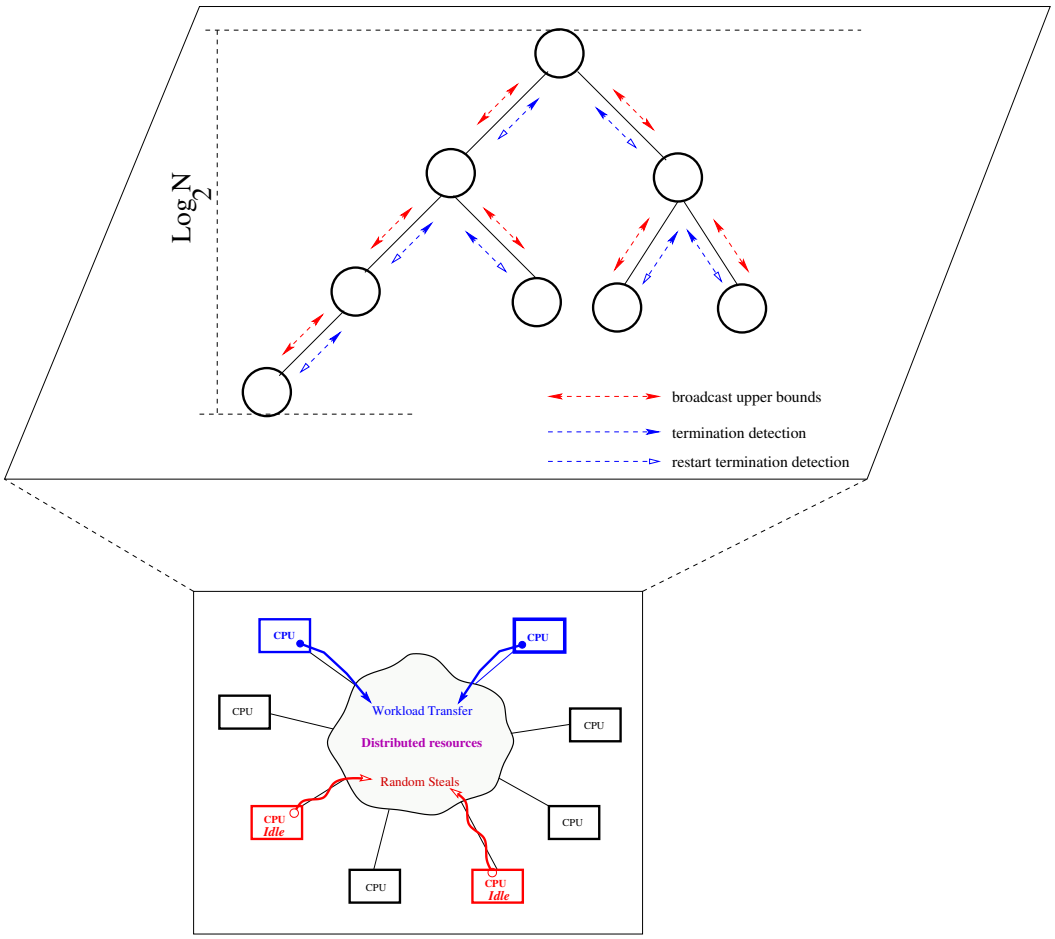

Figure 2.3: Communication for broadcasting a solution and termination detection

children. The process is recursively repeated until all PUs receive the new upper bound.

Compared to the simple broadcast mechanism, this hierarchical approach enables an efficient broadcasting. In fact, the straightforward broadcast use $1 - to - N$ communication and takes N iterations to broadcast a message to N available PUs. However it takes up to $2 * log_2 N$ iterations in the hierarchical approach.

2.3.3.2 Termination detection

One issue in the template of Algorithm 2 is how to detect termination distributively (Line 1). For B&B, this occurs when all tree nodes are explored (explicitly or implicitly, i.e., pruned). However, since stealing is performed locally by idle PUs, the work remaining in the system is not maintained anywhere. Normally, the termination detection runs in background and in parallel with other operations (e.g useful computation, load balancing, knowledge sharing)

To tackle this issue without introducing any complexity in the design, or any overhead causing performance degradation, we decided to use the fully distributed scheme adopted in the binary tree presented above. The termination is detected in an *'Up-Down'* distributed fashion. In the up phase, if a PU becomes idle and has not served any stealing request, it will then integrate a *positive* termination signal to its children signals. If a PU turns to idle and has served at least one stealing request, it will then integrate a *negative* termination signal to its children signals. Then the termination signal is forwarded to the parent and eventually to the root. In the down phase, if the root receives at least one *negative* termination signal from its children, it broadcasts a signal to restart a new round of termination detection. Otherwise, if only *positive* termination signals are received, the root broadcasts a message to announce global termination. The tree overlay used in our implementation allows a message between root and leaves to take at most $log_2 N$ hops. Therefore this enable us to scale out PUs while avoiding communication bottlenecks and performance degradation once a termination phase is performed.

2.4 Overlay-based Work Stealing and Parallel B&B

2.4.1 Preliminaries

Designing efficient load balancing protocols for parallel B&B is challenging. Technically speaking, there are two separate components in a B&B algorithms. The first component is responsible for load balancing and it is the core of the protocol. The second component handles other control operations like knowledge sharing and termination detection. The second component does not directly contribute to a good load balance or performance of a protocol, however it is still very important, especially for parallel B&B.

Actually, structuring computing nodes in a specific overlay shall allow us to address the issue of where to find work in a simple and efficient manner. In this section, we will present an overlay based protocol, namely a tree, which will cope with all aspects of B&B parallel computations: load balancing, knowledge sharing and termination detection. In the following, we shall the assume that computing nodes form a rooted tree where every node knows his parent and his children in the tree.

2.4.2 Tree-based work stealing

Let us assume that the target application to be parallelized is initially pushed at the root PU. Throughout the algorithm execution, an idle PU first chooses a target PU among its children in the tree to send a steal request asking for a piece of work. This is to contrast with RWS where victims are chosen uniformly at random. The strategy for PU selection plays a key role. In our distributed protocol, an idle PU steals downwards and upwards in the tree. In the down phase, every idle PU first requests its children. The steals are sent sequentially by choosing a child

uniformly at random at each step. Then, if and only if all children are idle, a steal is sent at last upwards to the parent. Notice that if a child has in parallel sent a request to its parent, then the parent needs not to steal from that child, thus saving some communication messages. Conceptually, this corresponds to a random work stealing strategy, but considering only the set of children that have not sent a request upwards yet. Algorithm 3 sketches this load balancing strategy. Notice that by *synchronous* request, we mean that a PU will wait until receiving a response from the corresponding neighbor in the tree (either children or parent).

Algorithm 3: TREE-BASED WORK STEALING

```
1 C ← set of my children PUs in the tree;
2 if All PUs from C already sent a work request then
3 |   u ← pick the parent PU;
4 |   Send a synchronous load balancing request to u;
5 else
6 |   u ← pick a PU that potentially has work from C;
7 |   Send a synchronous load balancing request to u;
```

The tree-based work stealing exhibits much locality, since tasks inside a subtree will always be completely finished before load balancing requests are sent to the parent of the subtree. This property of the protocol exposes some drawbacks. Firstly, a PU only steals upward when its subtree becomes idle. Therefore the whole subtree remains idle during the round trip of the steal request which may cause a significant lost in performance. The issue is more severe in large scale as the size of a subtree grows proportionally to the number of PUs in the system. Secondly, the performance of this protocol strongly depends on the tree diameter. If the tree diameter is large (i.e a PU has small number of PUs as its children), workload flows slowly from one side to another side of the tree (e.g binary tree). Otherwise, if the tree diameter is small, bottleneck could occur at some points in the tree (e.g star topology). Therefore, the tree diameter should be well optimized in order to produce good performance.

2.4.3 Bridge-based work stealing

The tree-based work stealing protocol described above suffers from the stalling of a set of PUs belonging to the subtree when stealing upwards. To tackle this issue, while maintaining the low-cost communication of the tree, we developed Bridge-based work stealing which is an extension of the above approach. This approach aims at speeding up work flow from overloaded subtrees to under-loaded ones. Apart from the tree edges, we propose to connect PUs being far away each other using *bridges*. Those bridge edges are to be viewed as logical shortcuts that can be traveled by work to reach under-loaded subtrees more quickly. Hence, bridge edges tend to minimize the dependency of our protocol on the tree diameter, thus leading to the

best achievable performance of the overlay.

In this approach, every PU v *further* requests work from *one* PU r through a bridge edge $b_{v\to r}$ chosen uniformly at random among PUs *being neither children nor parent*. More precisely, *in parallel while requesting its neighbors in the tree*, every idle PU v *asynchronously* sends a steal request over $b_{v\to r}$. Such an asynchronous steal request does not block v while waiting for a response from r. Instead, v is allowed to concurrently search for work from its neighboring PUs in the tree. If the remote neighbor r owns work, then it immediately services v. If r is idle, then this means that r has already sent an asynchronous work request through its bridge edge, and it is also requesting its respective direct neighbors. Thus, whenever an idle node, say p, gets work from its neighbors or through its bridge, then it immediately services all nodes from which a steal request was received. Let us remark that this distributed strategy operates in a recursive manner, implicitly building up a logical cluster of idle PUs. Consequently, all idle PUs are more likely to cooperate efficiently in searching for fresh work units. Algorithm 4 present the load balancing mechanism of this approach in more details.

Algorithm 4: Bridge-based work stealing

```
1  C ← set of my children PUs;
2  if All PUs from C already sent a work request then
3  |   u ← pick the parent PU;
4  |   Send a synchronous load balancing request to u;
5  else
6  |   u ← pick a PU that potentially has work from C;
7  |   Send a synchronous load balancing request to u;
8  if A remote request has not yet been issued then
9  |   r ← a remote PU corresponding to a bridge edge;
10 |   Send an asynchronous request to r;
```

Notice also that a PU could acquire more than one piece of work (from both a neighbor and a bridge), which we logically append to each other when computing the amount of work to send to other requesting peers. For the correctness of our bridge-augmented load balancer, one can also ask whether the asynchronous work requests may cause deadlock issues, e.g., u chooses r as its $b_{u\to r}$, simultaneously r chooses u as its $b_{r\to u}$ and neither u nor r could get new works. Since the designed overlay assures that if there is a place which has work, there is always a path from it to other places in the system, one can prove that this kind of issues can never happen, i.e., our protocol is correct and deadlock free.

2.4.4 Cooperative tree-dependent work sharing

The previous section described a stealing mechanism based on a tree overlay. It aims to solve the question how to choose a target PU to steal when a PU runs out

of works. It is in fact one of the crucial issue in dynamic load balancing. Another crucial issue in dynamic load balancing is the work sharing policy. This policy aims at distributing workload among idle and active PUs, therefore resulting in reducing idle time for stealing. A bad policy would in fact lead to the situation where many steal requests are performed thus inducing a loss in parallel efficiency.

In our approach, we propose to dynamically adjust the amount of work transferred from a PU to another according to the size of overlay subtrees. The main idea behind our protocol is based on the simple observation that idle PUs should not be selfish when searching for work, but should acquire enough work to serve their neighbors. More precisely, our work sharing policy is overlay-dependent: a PU divides its current work into the ratio of its own tree size and the tree size of the requesting PU as depicted in Algorithm 5. One should notice that each node must know the size of its own subtree and also the size of its parent subtree. This is computed in a fully distributed manner using a classical converge-cast process starting from leaf nodes until reaching the root.

Algorithm 5: Cooperative Work Sharing

```
   // PU u sends work to PU v
1  T_u ← tree size of u;
2  T_v ← tree size of v;
3  if v is parent of u then
4  |   work ← ((T_v − T_u) / T_v) · (current_works);
5  else if v is children of u then
6  |   work ← (T_v / T_u) · (current_works);
7  else if v is not neighbor of u then
8  |   work ← (T_v / (T_u + T_v)) · (current_works);
9  return work;
```

2.5 Application Benchmarks

In order to evaluate the performance of our dynamic load balancing protocols, we consider the Flow-Shop optimization problem as the benchmark application. Furthermore, to be more generic, we also evaluate our algorithms with the Unbalanced Tree Search application. In the next subsections, we will introduce these two applications in more detail and motivate their use in studying the performance of our load balancing algorithms.

2.5.1 Flow-Shop

Flow-Shop problems are NP-hard [Garey 1976] optimization problems. In this thesis, we consider the C_{max} Flowshop which consists in finding the optimal schedule of N jobs on M machines. Each job is scheduled only once on each machine without interruption and each machine can process only one job at a time. The set of jobs and machine are represented by $J = \{j_1, j_2, \cdots, j_N\}$ and $M = \{m_1, m_2, \cdots, m_m\}$ respectively. Each job j_i has a set of m operations $O_i = \{o_{i1}, o_{i2}, \cdots, o_{im}\}$ where o_{ik} is the operation of the job i executed on machine m_k for a duration d_{ik}. There are some constraints for this Flow-Shop problem. Firstly, the job j_i can not start the operation o_{ik} on machine m_k if the previous operation o_{ik-1} on machine m_{k-1} is not yet completed. Secondly, the sequence of jobs must be the same on every machine (e.g if j_1 is processed in the last position on the first machine, then it has to be executed in the last position on others). There are some variants of the Flow-Shop problems based on the objective criteria, however we only consider the C_{max} Flow-Shop problem in our work. The objective is to minimize the makespan, which is the elapsed time between the first job on the first job on the first machine and the last job on the last machine, when scheduling N jobs on M machines. Fig 2.4 presents an example of a problem of scheduling 3 jobs on 4 machines.

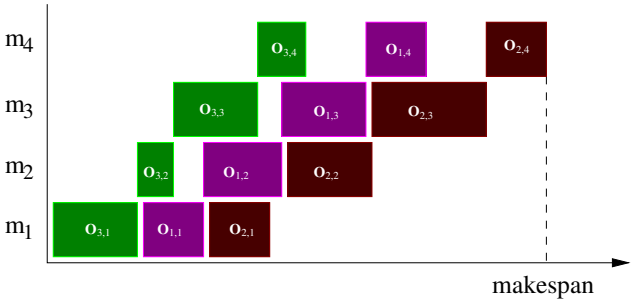

Figure 2.4: Example of flow-shop permutation problem

We consider the well-known Taillard's instances $\{Ta_{21}, \cdots, Ta_{30}\}$ of the family Ta-$20 * 20$, i.e., 20 jobs and 20 machines [Taillard 1993]. These instances are highly irregular compared to others and the sequential time needed to solve them ranges from some hours to some days, making them good candidates to experimentally validate dynamic load balancing algorithm in large scale environment. Let us notice that larger instances, e.g the family Ta-$50*50$, generate a very huge search consisting of excessive amount of works so that load balancing is relatively not a big issue.

2.5.2 Unbalanced Tree Search

Our works are at the frontier of optimization and HPC. To be more generic and not specific to B&B, we also take into account other application benchmarks coming from the HPC community. We decided to use the Unbalanced Tree Search (UTS) [Olivier 2007] benchmark in order to experimentally validate our protocols.

The UTS was specially designed to be *the* representative benchmark to evaluate the parallel performance of state space exploration and combinatorial search algorithms. Although UTS is relatively simple to design and experiment, it is a reference benchmark in HPC. In fact, it consists in an exhaustive parallel depth-first search to explore/count all nodes of a parameterized tree with extreme variation/imbalance in the relative size of its induced subtrees. The tree is constructed using a splittable, deterministic random stream generated using the SHA-1 secure hash algorithm. Each node is represented by a 20-byte SHA-1 digest and its children are found by applying the SHA-1 algorithm to the parent node's digest combined with the child id. There is a high degree of variation in the size of each subtree rooted at any given node in a UTS tree. Thus, if each node is taken as a task in a UTS execution there is a high degree of variation in the amount of work contained within each task. These properties make UTS be an excellent adversary benchmark application for dynamic load balancing schemes.

2.6 Large Scale Experimental Analysis

Our goal is to maximize the load balance with least communication overhead ultimately scaling up as much as possible and thus reducing the execution time of target application. Therefore we conducted extensive large scale experiments and extracted many measures to help understanding the properties of the designed protocols. In particular, three 'parameters' having a significant impact on our protocol are studied: (i) the structural properties of the underlying tree topology, (ii) the strategy used to share work, and (iii) the complexity/nature of the jobs being processed in parallel. We shall also focus on the scalability of experimented protocols and their behavior in terms of load balancing. Besides, we also consider other algorithms (i.e Master-Worker (MW) [Mezmaz 2007c], Adaptive Hierarchical Master-Worker (AHMW) [Bendjoudi 2012b] and the Random Work Stealing (RWS)) and compare them with our tree-based approaches in order to fairly elicit the relative performance of the different available protocols.

2.6.1 Experimental setting

Two clusters C_1 and C_2 of the Grid'5000 [Grid'5000] were involved in our experiments. Cluster C_1 (resp. C_2) has 92 nodes (resp. 144 nodes), each one equipped with 2 CPU of 2.5 Ghz Intel Xeon processor with 4 cores per CPU (resp. 1 CPU of 2.6 Ghz Intel Xeon processor having 4 cores) and a network card Infiniband-20G. Once some nodes of clusters C_1 or C_2 are reserved through the Grid'5000 reservation system, they are exclusively owned by the user, but the network is not completely dedicated to that user. For a scale of $n < 800$ cores, we use cores of cluster C_1. For a scale of $n \geq 800$, we use cores from both C_1 and C_2.

In the remainder, we use notation $\mathrm{T_R}$ to refer to our distributed protocol running over a *randomized tree* overlay constructed as following. Starting with the first node as root, children nodes are chosen uniformly at random in the already constructed

tree. We use T_D to refer to a *deterministic tree* overlay with a fixed upper bound d_{max} on the maximum number of children per node. More precisely, depending on the number of peers, the overlay tree is constructed starting with a root node and packing at most d_{max} nodes in the first level. Then, we loop over the nodes of the new level packing again at most d_{max} children per node, and so on. We use BT_D to refer to the extended version of T_D (see Subsection 2.4.3), where each node in BT_D further chooses a *random bridge edge* to ask in parallel for work. If not stated explicitly, our overlay based work sharing is always adopted when computing the amount of work to transfer between nodes.

In the following experiments, all the protocols are deployed before application execution, where each core plays the role of a PU. No specific binding between peers and cores is adopted, i.e., PUs are just thrown randomly on available cores. After completely built, the application (UTS or B&B) is pushed into an initial PU, i.e., the root PU in case of our approach, MW and AHMW, a random PU in case of RWS, to start the parallel computation phase. The deployment phase is described and evaluated in the next subsection.

2.6.2 Comparison between parallel and sequential deployment

Deploying efficiently distributed applications in large scale environments is critical since the cost of a deployment could be very expensive, especially in large scale systems [Bendjoudi 2012b]. The deployment includes initializing PUs on available CPU cores, constructing necessary overlay to connect PUs (e.g binary tree for knowledge sharing and termination detection in RWS, star topology in MW, or a tree in our approach), and pushing the problem to be solved into the constructed PUs. The critical point in this process is the cost of initializing and starting PUs on available CPU cores.

In this section, we consider two deployment techniques: sequential and parallel. The sequential one is the basic and simplest strategy consisting in launching PUs one by one on a list of available CPU cores. It is obvious that despite the simplicity this approach leads to a high deployment cost which may exceed the time needed to run our applications. Therefore, we designed a parallel strategy inspired by collective communication procedures in standard message passing libraries. More precisely, depending on the number of PUs we want to deploy, we proceed in a recursive manner; where each CPU core hosting a newly deployed peer is in turn responsible for starting new PUs on other cores *in parallel*, and so on until all PUs are initialized. It is not difficult to see that using a tree structure where each core is responsible for starting its children, the number of steps required to start n PUs is $O(\log(n))$ which is to contrast with the $O(n)$ steps required in the sequential one. Figure 2.5 presents an analysis of the cost of the two deployment techniques presented above.

To assess the performance of this strategy in practice, Fig. 2.6 shows the time taken to deploy PUs in function of n for both the simple sequential deployment strategy and the parallel one. We can clearly see that, as the number of PUs increases, the deployment cost stays relatively very low in the parallel deployment strategy,

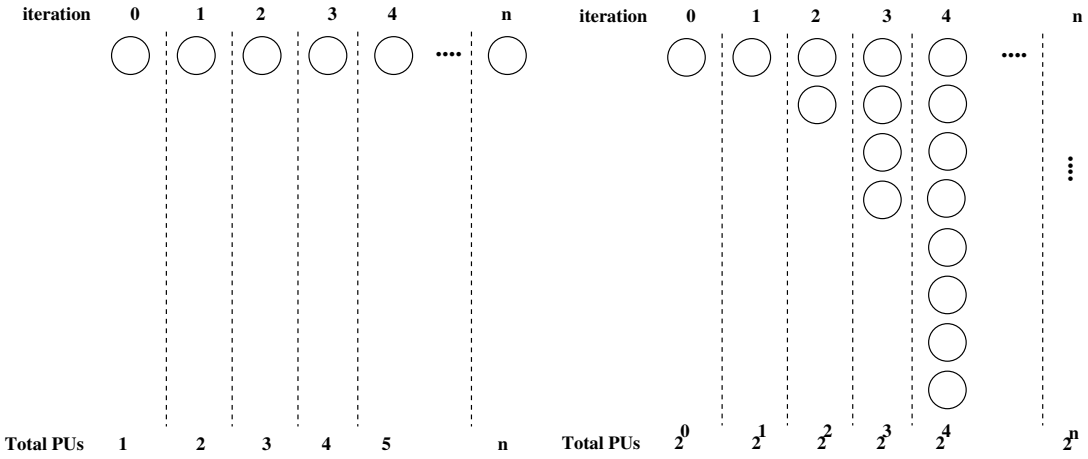

Figure 2.5: Analysis of deployment cost. **Left**: Sequential Deployment. **Right**: Parallel Deployment

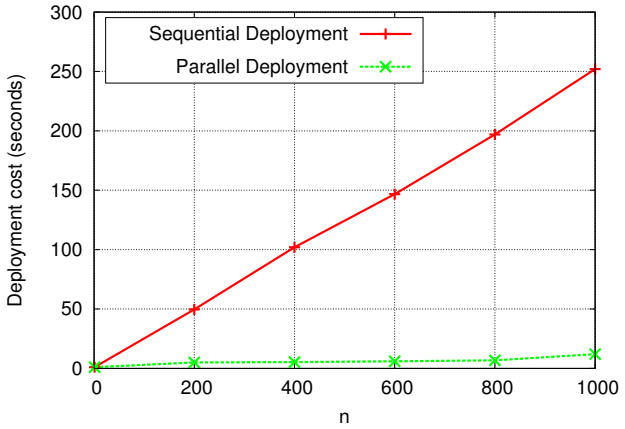

Figure 2.6: Deployment cost as a function of size n

whereas it increases linearly for the sequential strategy and it is not negligible (with respect to the time it takes to run the whole application as it can be observed later in this). Therefore, the parallel approach is chosen for the deployment phase of all experiments presented in this chapter as well as in the following ones.

2.6.3 Comparison between T_R and T_D

2.6.3.1 Impact of tree structure

Let us consider some benchmark instances of B&B and UTS and let us study the relative performance of T_D and T_R under different overlay configurations. Results are summarized in Fig 2.7. We see that the execution time of the distributed protocol

highly depends on the shape of the tree overlay. For a deterministic tree, execution time decreases as we increase the degree. Overall, a deterministic tree performs better compared to a randomized one (T_R). We also observe that as we increase d_{max}, the protocol becomes more stable. In fact, as the degree of the tree increases, the distance between computing nodes decreases, thus making workload flows more quickly.

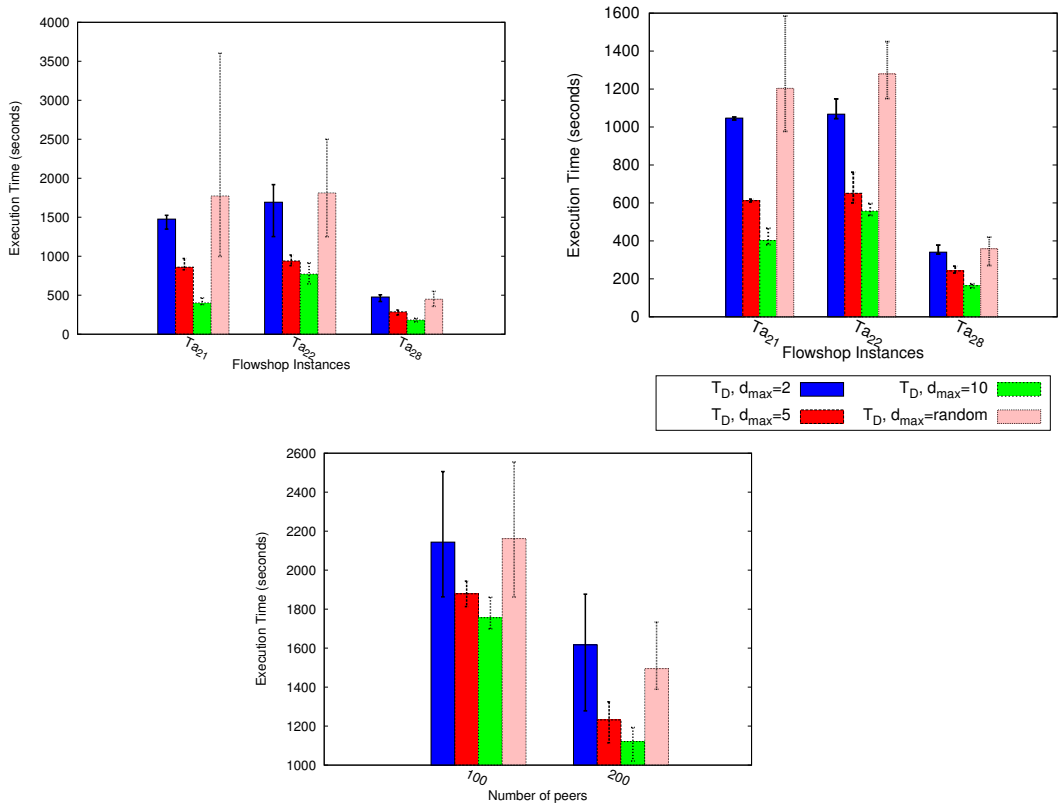

Figure 2.7: Impact of tree degree and shape. **Top Left** (resp. **Top Right**): B&B at scale 100 cores (resp. at scale 200 cores) averaged over 10 runs and using Flowshop instances Ta_{21}, Ta_{22} and Ta_{28}. **Bottom**: a UTS benchmark with Binomial distribution of size 157 billion nodes, i.e., generator parameters: (b=2000 q=0.4999995 m=2 r=599).

To fully understand the impact of tree degree and diameter on execution time, load distribution and any congestion in the network, we conduct a second set of experiments at the higher scale of 500 cores. One can clearly see (Fig. 2.8 Top) that by increasing tree degree we gain in execution time, but quickly the gain becomes marginal as we increase the degree beyond some threshold (around 6). In fact, workload flows faster for large degree since the distance between nodes is minimized. However, this has a price, as confirmed by the distribution of message requests over tree nodes (Fig. 2.8 Bottom Left and Bottom Right). Although execution time

tends to decrease for larger degrees, communication load gets higher at intermediate nodes, i.e. message traffic is mostly supported by non-leaf tree nodes, thus inducing communication delays at those nodes.

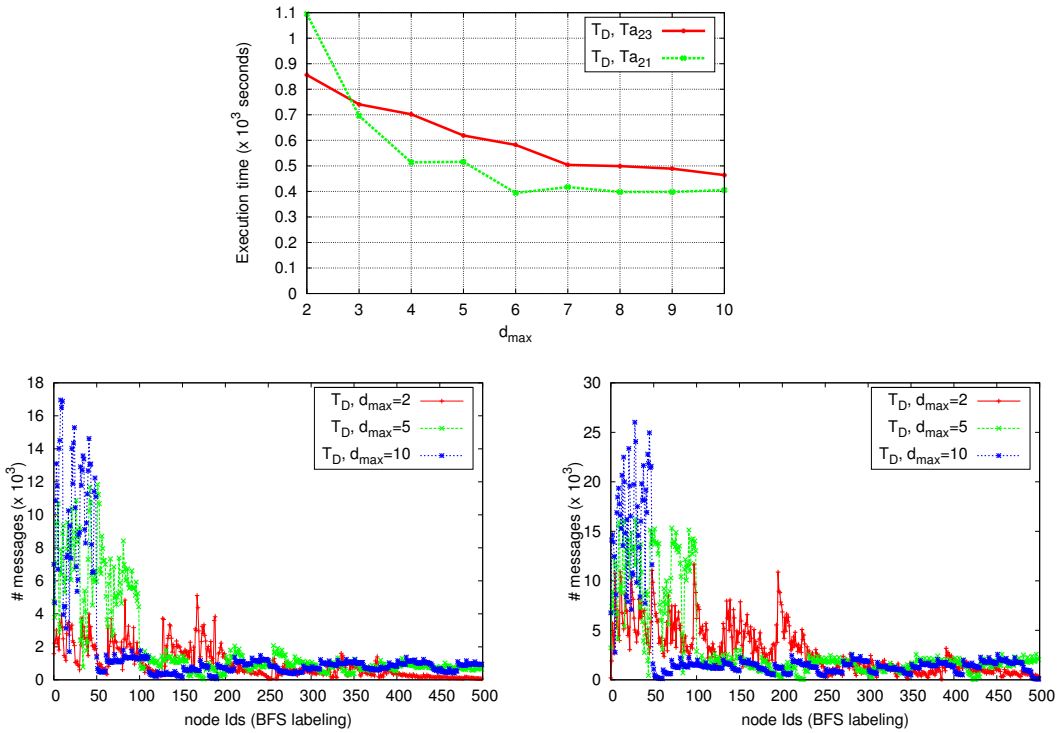

Figure 2.8: **Top**: Execution time using 500 cores as a function of $d_{\max}$ for two B&B instances (Ta_{21} and Ta_{23}). **Bottom Left** resp. **Bottom Right**: Number of messages sent by each peer respectively for instance Ta_{21} and Ta_{23}, and different tree degrees. The x-axis refers to peers identifiers where peers are numbered in a BFS manner, i.e., with $d_{\max} = 10$ (resp. 5) the root has id 0, peers in the first level have ids 1 to 10 (resp. 1 to 5), and so on.

2.6.3.2 Impact of work granularity policy

To study the performance of our strategy compared to the situation where the amount of transferred work is fixed by a parameter, we consider the widely used strategy of dividing work in two halves. In Fig. 2.9, we report execution time obtained for ten B&B instances at a scale of 200 cores and for UTS up to a scale of 128 cores. One can clearly see that our subtree proportional work load distribution performs substantially better than the steal-half strategy, independently of B&B, UTS and network scale. In Fig. 2.9 Top Right, we draw the total number of work requests injected to the network by both strategies. We can clearly see that execution time and work requests are perfectly correlated. Thus, we can say that the overlay propor-

tional strategy tends to guide the load balancing operations in order to result in the best performance. Recall that too few or too many load balancing operations cause the situation from which the performance can fall down [Olivier 2007, Olivier 2008].

We conclude this section by remarking that although the ten B&B Flowshop instances have the same theoretical size (20 jobs on 20 machines), their effective complexity may vary substantially making some instances harder to solve than the others. This can be attributed to two facts: (i) depending on the instance, B&B is able prune a variable number of branches, and (ii) work distribution implies starting exploring one region in the solution space before another one which can impact the best found solution upper bound, thus the number of explored branches and consequently execution time. This empirical claim shall be verified in the next section where the best variant of T_D ($d_{max} = 10$) is compared with BT_D.

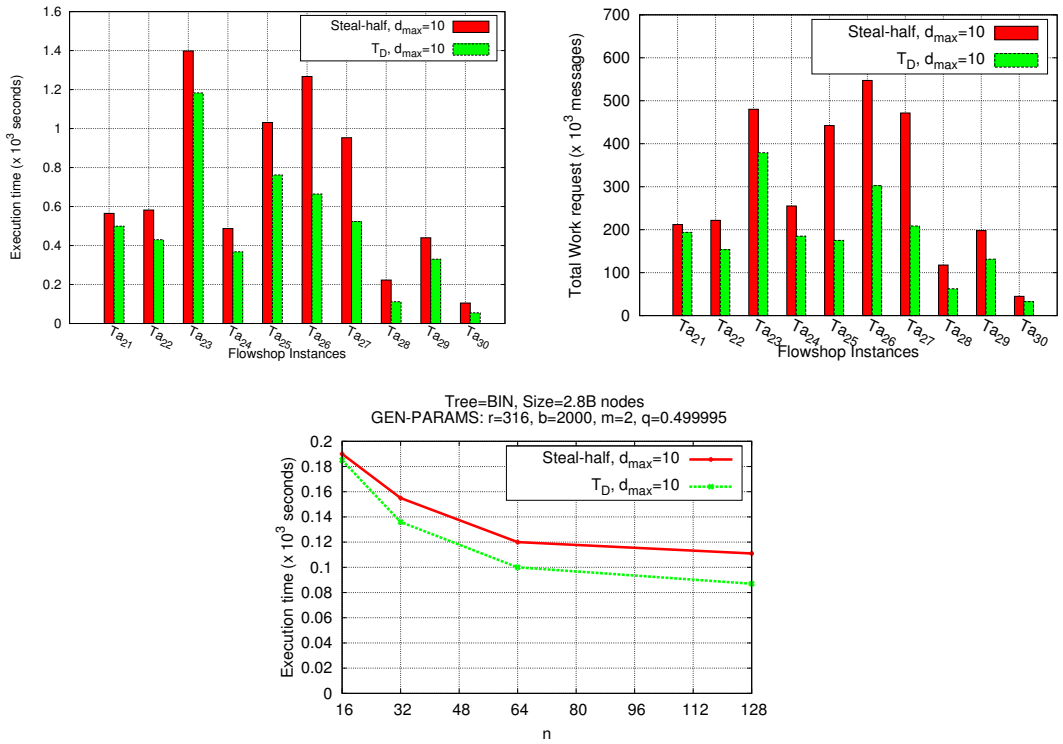

Figure 2.9: Comparison of overlay-based work transfer and steal-half: **Top Left** (resp. **Top Right**) : Execution time (resp. number of work requests) using 200 cores for the 10 B&B instances. **Bottom**: Execution time for UTS as a function of overlay size n.

2.6.4 Comparison between T_D and BT_D

In this section, we study the effect of adding bridge edges to the overlay tree by comparing T_D and BT_D. In the first column of Fig. 2.10, we report the execution

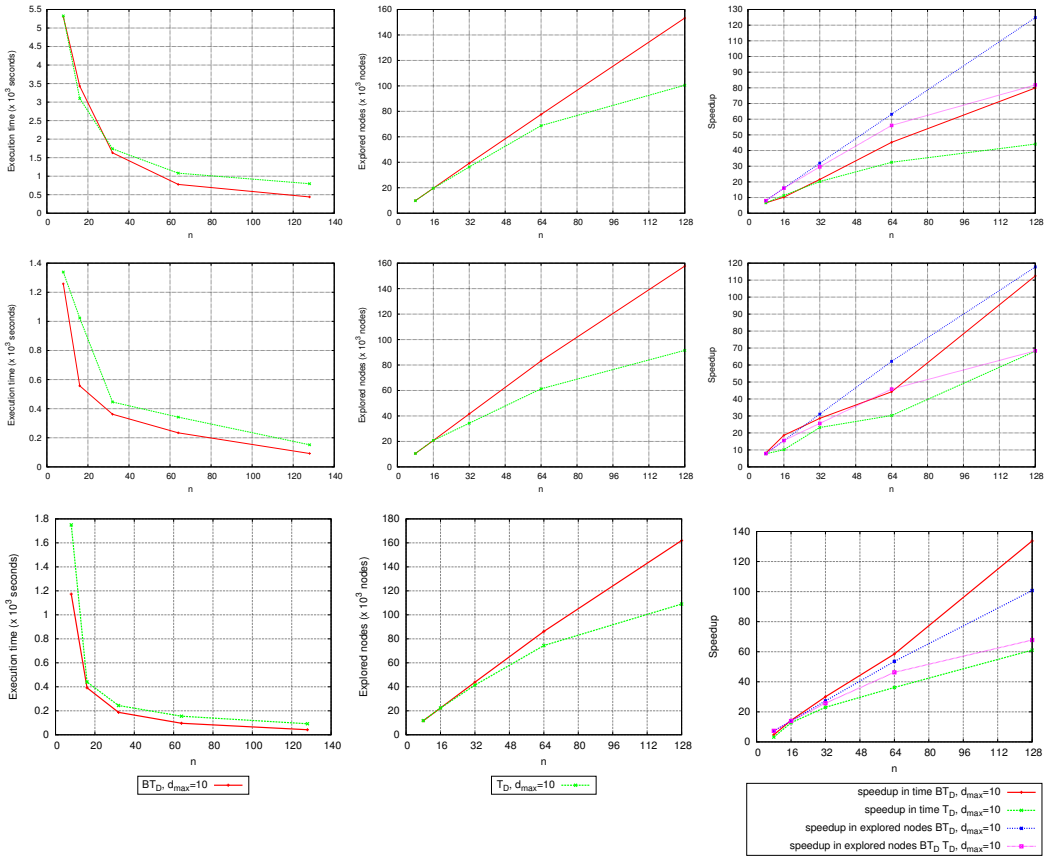

Figure 2.10: Results of T_D vs. BT_D for three B&B instances: Ta_{21} (first row), Ta_{28} (second row) and Ta_{30} (third row). **Left column**: Execution Time. **Middle Column**: Number of explored nodes per second. **Right** Speedup with respect to a sequential execution.

time we obtain for three different B&B instances and at a scale ranging from 8 to 128. The first observation is that BT_D outperforms T_D. This confirms the fact that bridge edges help the work to flow quickly over the tree, thus minimizing the number of idle nodes and improving performance. In fact, in Fig. 2.11 we depicted the 'life-story' of a run at scale 128 for the three considered instances by distinguishing between nodes which are in a compute state (i.e., processing work) and those which are idle (i.e., requesting for work or sharing work). To obtain those results, we instrumented our application for each computing node and recorded every fixed time delay the state of the corresponding node (either computing or idle). The start time for these life-stories was normalized to 0 to take into account clock skew between the nodes. The histories were then merged in an order preserving way to determine the composite picture presented in Fig. 2.11. To interpret the picture, one can focus on the size of the green area, i.e., the smaller it is, the better the performance. One can clearly see that while BT_D is almost optimal, the number of idle nodes is not negligible with T_D independently of the considered B&B instance.

In Fig. 2.10 (second column), we also show the performance in terms of number of B&B nodes explored, i.e., the number of branches effectively explored overall all computing cores. This allows us to account for the irregularity of the B&B search process and to better illustrate the load balancing of our protocol independently of the tackled instance. In fact, since sharing the work differently could result in discovering different solutions more or less quickly, this can have a deep impact in the elimination (pruning) operator of the B&B search process. Thus, even when considering the same B&B instance, this can result in different amount of B&B search nodes to explore in parallel depending on the scale or on the work sharing strategy. The number of B&B search nodes provides a more uniform measure informing how much work was generated overall the execution. We can see that although T_D generates less work overall, it cannot outperform BT_D which can only be attributed the efficiency of the load balancing mechanism. In addition, we can remark that the speed-up in terms of number of explored nodes per second is linear for BT_D (third column in Fig. 2.10). The speed-up in execution time is worst, but BT_D is still able to scale efficiently. Notice also that the speed-up in execution time is super-linear at scale 128 for instance Ta_{30}. This may appear surprising at a first sight. However, by the discussion made above this can be explained by the fact that at this scale a very good solution (actually the optimal one) is found very quickly so that the pruning is very effective and the parallel B&B is very efficient. We emphasis the fact that in order to fully appreciate the speed-up of our approach as we increase the scale, one should keep in mind that the size of the induced B&B search process may not remain constant.

2.6.5 T_D and BT_D vs. AHMW for B&B

In this section, we compare our approach with the so-called adaptive hierarchical master-worker (AHMW) approach studied very recently in [Bendjoudi 2012b, Bendjoudi 2012c], specifically for B&B. AHMW is mostly related to our work since

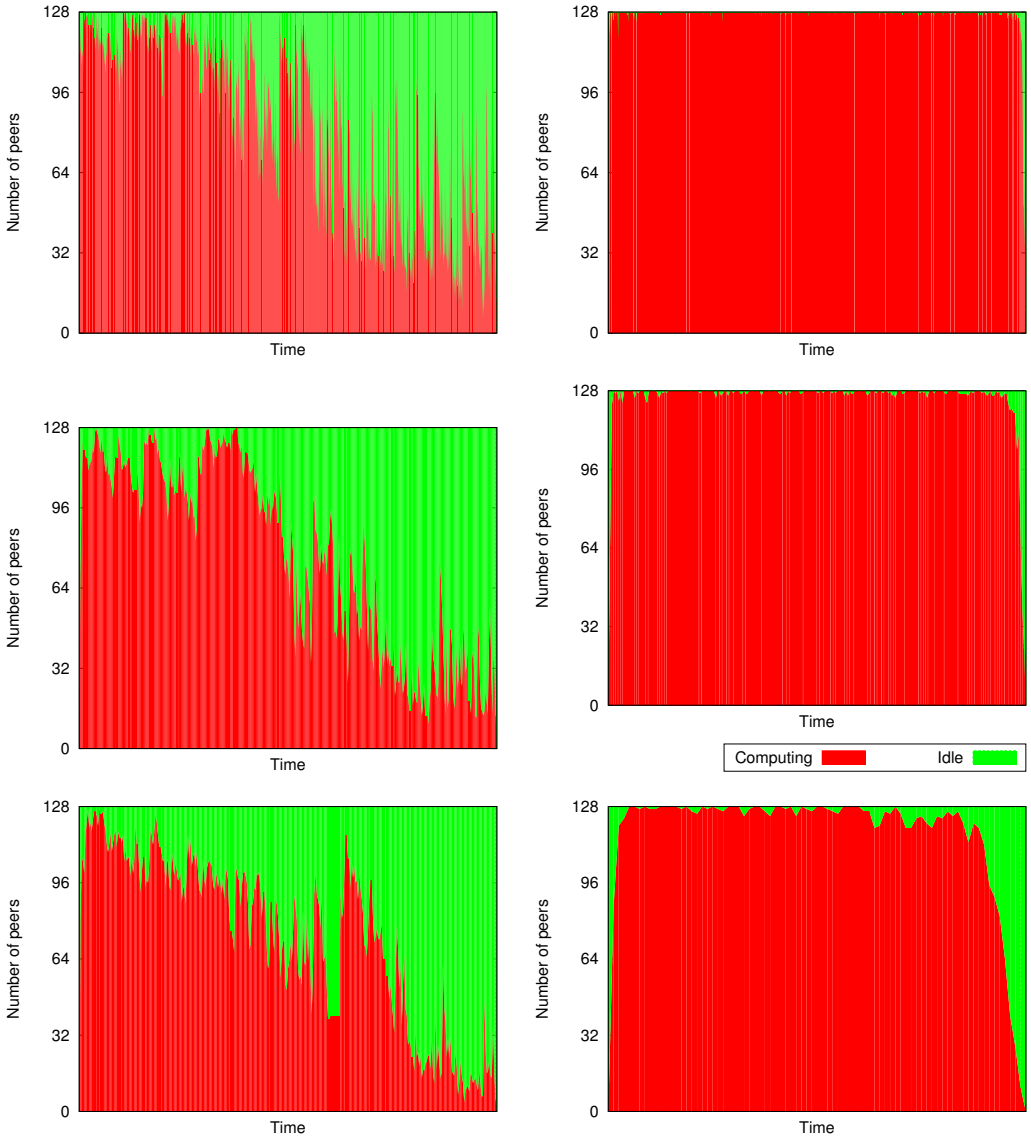

Figure 2.11: 'Lifestory': Peer status over time in scale of 128 cores of T_D and BT_D for three B&B instances Ta_{21} (first row), Ta_{28} (second row) and Ta_{30} (third row). **Left**: T_D. **Right**: BT_D.

.

it explicitly organizes computing nodes in a tree hierarchy, and then adapts the computations according to that tree. For this chapter to be self-contained we recall the design principles and distributed policies used for AHMW. We first notice that the parallel B&B-specific *search algorithm* induced by AHMW is conceptually different from ours in all aspects.

In AHMW, computing nodes are organized in a hierarchical topology, thus inducing a tree backbone. Every node can both play the role of a master and/or a worker, depending on its height in the hierarchy. Furthermore, masters belonging to the same hierarchy level can directly communicate and share work with each other. The global B&B search tree is then decomposed into B&B subtrees which are mapped into the master hierarchy dynamically at runtime. In fact, the general idea of AHMW is to adapt the size of the B&B sub-trees being processed by each master/worker in an attempt to balance the load evenly. Roughly speaking, each master owns a work pool corresponding to sub-problems partially explored in its corresponding B&B sub-tree. When the work pool becomes empty, a master steals a sub-problem from its parent. It then re-generates a new work pool, and so on. B&B work grain plays a crucial role in AHMW. It corresponds to the depth at which a master/worker is allowed to explore a sub-problem. It is tuned to be a function of every master level in the overlay hierarchy which is shown to allow efficient and adaptive B&B work distribution among masters. (Notice that AHMW [Bendjoudi 2012b, Bendjoudi 2012c] is argued to perform best when the tree hierarchy has degree 10, which is in a way consistent with our study). The results obtained with AHMW at scale of 200 cores in comparison with our approach for configurations $\mathrm{T_D}$, $\mathrm{BT_D}$ and $d_{max} = 10$, are summarized in Table 2.1.

	$\mathrm{T_D},\ d_{\max} = 10$	$\mathrm{BT_D},\ d_{\max} = 10$	$AHMW$	$\frac{AHMW}{T_D}$	$\frac{AHMW}{BT_D}$
$\mathrm{Ta_{21}}$	**499**	**354**	15804	31.6	44.6
$\mathrm{Ta_{22}}$	**430**	**224**	438	1.01	1.95
$\mathrm{Ta_{23}}$	1183	791	776	0.656	0.98
$\mathrm{Ta_{24}}$	**368**	**194**	3352	9.11	17.27
$\mathrm{Ta_{25}}$	**762**	**404**	2652	3.48	6.56
$\mathrm{Ta_{26}}$	**664**	**472**	3231	4.87	6.85
$\mathrm{Ta_{27}}$	523	**346**	445	0.85	1.28
$\mathrm{Ta_{28}}$	**112**	**65**	1208	10.79	18.58
$\mathrm{Ta_{29}}$	330	**68**	325	0.98	4.78
$\mathrm{Ta_{30}}$	**55**	**29**	303	5.5	10.49

Table 2.1: Execution time (in seconds) of T_D and BT_D compared with AHMW at scale of 200 cores. Bold style refers to execution time better than AHMW.

When considering the $\mathrm{T_D}$ strategy, we perform substantially better than AHMW

for 7 out of 10 instances. Using BT_D, our approach performs better than AHMW for 9 out of 10 instances. One can clearly see the relatively huge gap between AHMW and our approach, e.g., over all instances, BT_D (resp. T_D) is approximately 10 (resp. 5) times faster than AHMW. This set of experiments also shows that BT_D performs significantly better than T_D. This is naturally attributed to the bridge edges which are fully playing their role of speeding up workflow through the tree.

2.6.6 BT_D vs. MW vs. RWS for B&B

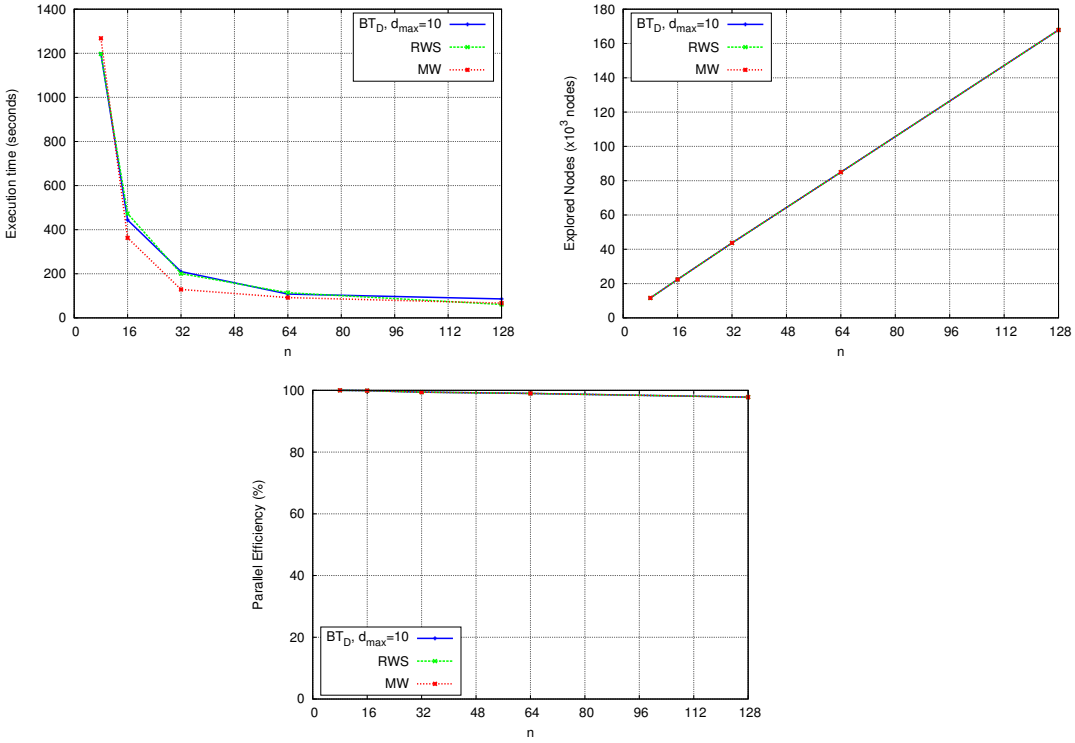

Figure 2.12: Results of BT_D, RWS and MW for B&B instance Ta_{30} at small scales. **Top Left**: Execution time. **Top Right**: Number of explored nodes per second. **Bottom**: Parallel efficiency.

To fully appreciate the performance of our approach, we further compare it to the Master Worker (MW) approach studied in [Mezmaz 2007c] and the well-known Random Work Stealing (RWS). For completeness, the important implementation issues raised by the MW approach is sketched in the following. The load-balancing and work distribution operations of this approach are fundamentally different from our approach. In particular, they are tuned to take into account specific properties of B&B works. This makes the MW approach an interesting candidate for evaluating the performance of our *generic* load-balancing scheme. In MW, there is a unique master playing the role of managing a global work pool for workers. The work

pool at the master consists of a set of unprocessed B&B and their corresponding workers. Workers periodically communicate with their master in order to update those B&B subproblems which have already been completed, and to acquire fresh work whenever their local work pool becomes empty. Whenever a master is asked for work, it shares a set of B&B subproblems to the requesting worker. Because the master could assign works *which are not completely disjoint* to different workers, some kind of redundancy may appear when executing the B&B in parallel. This issue is well studied in [Mezmaz 2007c] and was shown to have very negligible impact on overall performance, i.e., the redundancy reported in [Mezmaz 2007c] is of only 0.39% (in terms of B&B explored nodes).

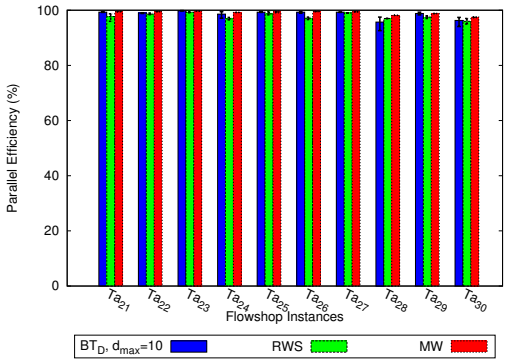

Figure 2.13: Parallel Efficiency of BT_D, RWS and MW for B&B instances at scale of 200 cores

In Fig. 2.12, we show the scalability of the three approaches BT_D, MW, and RWS while using up to 128 cores and B&B flowshop instance Ta_{30}. The three approaches appear to be comparable both in terms of execution time, number of explored nodes, and parallel efficiency (i.e., the ratio of computing time over execution time). Notice that the parallel efficiency is almost optimal which witness the effectiveness of the three approaches. The same observation can also be made at the larger scale of 200 cores as depicted in Fig. 2.13 summarizing the parallel efficiency of the three approaches for the 10 B&B instances. Particularly, although the MW may seem rather simplistic at a first side, it is actually very competitive against our approach and RWS which is a reference approach for dynamic load balancing. This can be attributed to two facts: (i) The MW approach of [Mezmaz 2007c] is well tuned to perform efficiently for B&B, and (ii) Such a centralized approach, where all dependencies are concentrated at a single point (the master), works well at a relatively low network scale. In next section, the relative scalability of the three approaches is analyzed in details for both B&B and UTS, but at the larger scales.

2.6.7 Scalability of BT_D vs. MW for B&B

In Fig 2.14, we compare our approach with MW at the scale of 1200 cores. One can clearly see that our approach is substantially better than MW for all instances, but

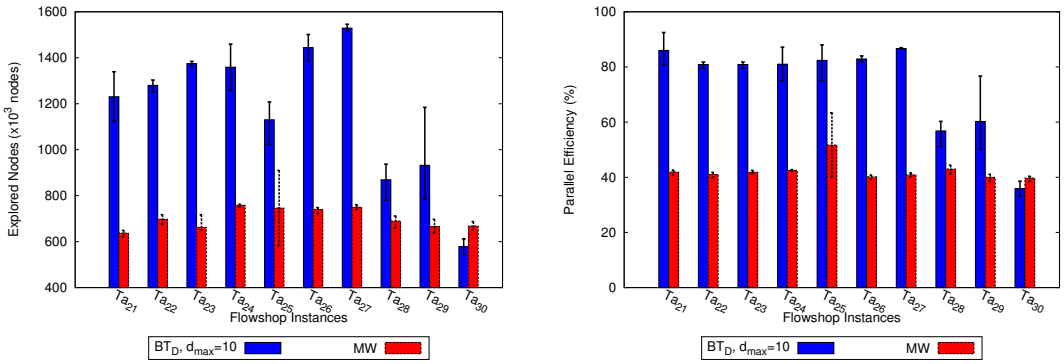

Figure 2.14: Scalability results of BT_D vs. MW for B&B. **Left** (resp. **Right**): number of explored nodes per second (resp. parallel efficiency), at scale of 1200 cores.

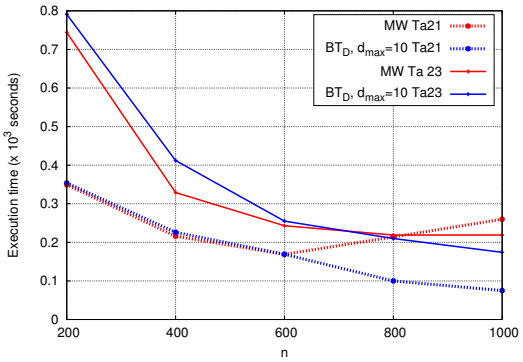

Figure 2.15: Scalability results of BT_D vs. MW for B&B: execution time of BT_D vs. MW as a function of overlay size n for instances Ta_{21} and Ta_{23}.

for Ta_{30} where both approaches are comparable. In Fig. 2.15, we further consider two instances Ta_{21} and Ta_{23} and we scale the network from 200 up to 1000 cores. As it can be observed, the performance of MW starts to slow down while scaling up. Specifically, when using more than 600 cores, the execution time for Ta_{21} starts to increase rapidly and the execution time for Ta_{23} decreases very marginally. This is attributed to the severe communication bottleneck at the master caused by fine-grain works. This contrasts with our BT_D scheme which is fully distributed so that it continues scaling for both Ta_{21} and Ta_{23} while efficiently distributing communication load with fine-grain works.

2.6.8 Scalability of BT_D vs. RWS for B&B and UTS

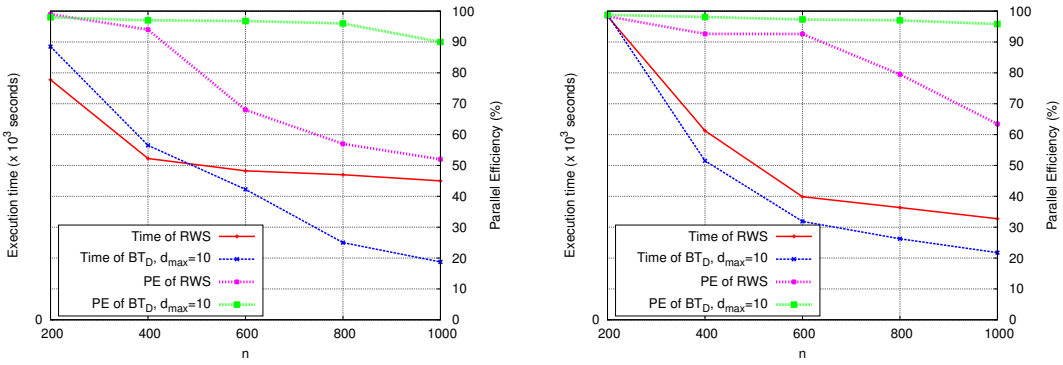

Figure 2.16: Scalability results of BT_D vs. RWS for B&B. Execution time and Parallel efficiency (PE), as a function of overlay size n, for B&B instances Ta_{21} (**Left**), Ta_{23} (**Right**).

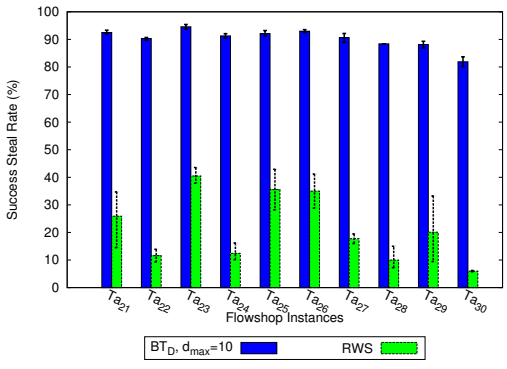

Figure 2.17: Scalability results of BT_D vs. RWS for B&B. average successful stealing rate of BT_D and RWS for the 10 B&B instances at scale of 1200 cores.

In Fig. 2.16, we report the execution time and the corresponding parallel effi-

ciency of BT_D against RWS for the two instances Ta_{21} and Ta_{23}. One can clearly see that RWS stays competitive up to 400 cores, but then it deteriorates dramatically compared with BT_D. Specifically, while the parallel efficiency of BT_D decreases marginally and stays above 90% (resp. 96%) for Ta_{21} (resp. Ta_{23}) in the scale of 1000 cores, it drops down quickly for RWS reaching 52% (resp. 63%) for Ta_{21} (resp. Ta_{23}). To better understand this performance drop, we report in Fig. 2.17 the ratio of successful steals, i.e., the proportion of work requests that allow a core to acquire work. One can clearly see that finding work becomes very difficult for RWS, i.e., only relatively few work requests are successful and are able to fetch work. As a consequence the load is not balanced evenly and the performance is getting down at large scales. The relative scalability of BT_D is confirmed when executed for the UTS benchmark as shown in Fig. 2.18. The parallel efficiency of BT_D is in fact substantially better than RWS, i.e, 77% vs 64%, using 512 cores.

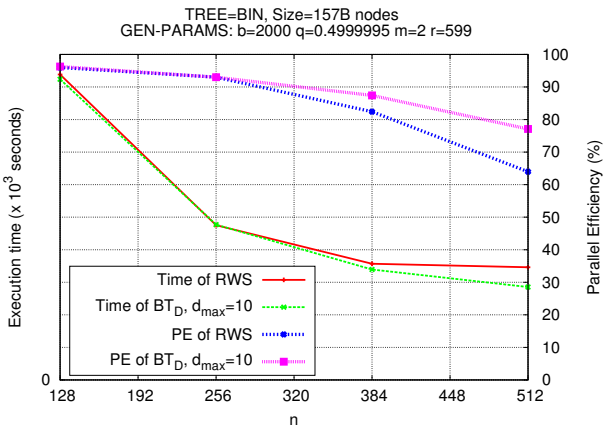

Figure 2.18: Results of BT_D vs RWS for UTS. Execution time and Parallel efficiency (PE), as a function of overlay size n for UTS.

Several conclusions can be drawn from this set of experiments. For relatively low network scales, RWS is confirmed to be very competitive which is consistent with previous studies. However, there is still an opportunity for further improvements as demonstrated by our BT_D scheme. By extending tree paths with bridge edges, we allow work to flow more quickly improving on RWS. At larger scales, RWS reaches its limits, since idle nodes try to catch victims 'blindly' using random requests over a fully connected overlay. In contrast, our tree centric approach tends to minimize communication delays by distributing the load in a more deterministic/cooperative manner and the gain in parallel efficiency, thus in speed-up, becomes substantial as we scale up the network.

2.7 Conclusion

In this chapter, we attempted to tackle the irregularity of parallel B&B generated at runtime during execution. Firstly, we presented and discussed briefly an overview of load balancing techniques. We then proposed to use the random work stealing to solve the irregularity issues. Secondly, we proposed and studied a new tree-based load balancing protocol for solving the irregularity in parallel B&B. In particular, we map all the available computing resources to a tree overlay in order to make them cooperate efficiently. Thirdly, we proposed a work sharing policy based on the subtree size of thieves and victims while stealing works along the tree. Instead of only stealing works for itself, a thief should try to steal works for its whole subtree, as later, it must serve the stealing requests coming from its children. Similarly, when a victim shares works to a thief, it takes into consideration the subtree size of the thief and itself in order to balance evenly the workload between the two subtrees. We also proposed to extend the tree by adding some bridge edges to make work flow faster in the distributed system. Finally, we conducted an extensive set of experiments to evaluate our tree-based solution as well as the random work stealing in several scenarios at large scales up to 1200 cores for both B&B and UTS. We also compare them with previous protocol, namely, Master-Worker and Adaptive Hierarchical Master-Worker. Through our extensive experiments, we observed that:

- The tree-dependent work sharing policy outperforms the standard steal-half one where a victim simply shares half of its work to a requesting thief. In fact, the steal-half uses more steal requests and spends more time in communication than our approach for balancing the workload.
- The structure of the tree overlay strongly impacts performance. If the tree overlay has small degree or large diameter, a good performance is hardly achieved as works must take more hops to flow from one side to another side of the tree. Moreover, our bridge-based work stealing outperforms the tree-based as bridge edges help the work to flow quickly over the tree, thus minimizing the communication time and improving performance.
- The Master-Worker or Adaptive Hierarchical Master have several shortcomings, thus producing poor performance and our approach is able to significantly improve their performance.
- The Random Work Stealing produces good performance for both B&B and UTS but it also struggles to balance the workload in very large scales as the applications are relatively fine-grained.

In summary, this chapter presented some load balancing technologies to handle the irregularity of B&B and UTS application. The studies were performed on homogeneous computing platforms where the compute nodes and interconnect networks are assumed to have the same characteristics and performance. However, most of computing platforms are heterogeneous in both compute nodes and interconnection networks. The next chapters will mainly focus on these heterogeneity issues.

Chapter 3

Dynamic Load Balancing for *Node-Heterogeneous* Computing Environments

Contents

Main publication related to this chapter

T. Vu, B. Derbel, and N. Melab. Adaptive Dynamic Load Balancing in Heterogeneous Multiple GPUs-CPUs Distributed Setting: Case Study of B&B Tree Search. *7th International Conference on Learning and Intelligent Optimization, LION 2013*, pages 87-103 of *Lecture Notes in Computer Science*, 2013.

T. Vu and B. Derbel. Parallel Branch-and-Bound in Multi-core Multi-CPU Multi-GPU Heterogeneous Environments. *INRIA Research Report*, 2014.

3.1 Introduction

Node-heterogeneous computing systems, interconnecting several hundreds of processing units (PUs) ranging from multi-core CPUs, multi CPUs to multi-GPUs, are the current trend in high performance computing. They provide an impressive computing power for solving large and challenging problems. Although the aggregated computing capability of those resources is very powerful in theory, achieving high performance and scalability is still bound to the expertise of programmers in developing new parallel techniques and paradigms operating both at the algorithmic and the system levels. The heterogeneity of resources in terms of computing power and programming models, make it difficult to parallelize a given application without significantly drifting away from the optimal and theoretically attainable performance. In particular, when parallelizing highly irregular applications producing unpredictable workload *at runtime* like parallel B&B, mapping dynamically generated tasks into the hardware so that workload is distributed evenly is a challenging issue.

In this chapter, we push forward the design of parallel and distributed B&B algorithms, in order to run them efficiently on node-heterogenous systems where a PU can be either single CPU, single CPU equipped with GPU, multi-core CPUs or multi-core CPUs equipped with GPUs. Given that all PUs coming from possibly different clusters connected through a network can be used to parallelize the B&B tree search, three major issues are addressed:

Q1. Can we benefit from the different degrees of parallelism available in the tree search procedure and map them efficiently into the different PUs?

Q2. Given no knowledge about the amount of work the search would produce, can we distributively coordinate PUs so that parallelism dynamically unfolds, while communication cost and idle time of PUs are kept minimal?

Q3. Having PUs with different computing abilities, can we distribute the load evenly in order to attain optimal speedup while scaling the network?

In the next sections, we answer the three questions in a positive manner while giving new insights into how to fully benefit from node-heterogenous computing systems and tackle irregularity issues of parallel B&B. The remainder of this chapter is organized as following. Section 3.2 briefly discusses our solution to deal with the three above questions. Section 3.3 presents the design and implementation of our algorithmic solution while using Multi-CPUs Multi-GPUs as computing platforms. Section 3.4 introduces an improvement of the previous design while adding the usage of Multi-cores into the computing platform. Section 3.5 reports our experimental results and findings. Finally, Section 3.6 concludes the chapter.

3.2 A comprehensive overview of our approach

In this section, we give the general principles guiding our approach which will be described later in detail. The goal is to introduce the different components of our approach in a comprehensive manner without going into system technicalities or implementation details. A thorough detailed discussion is presented below in order to answer the three questions of the previous section.

3.2.1 Mapping B&B Parallelism (Q1)

On one side, considering node-heterogeneous computing platform having different levels of hierarchy and computing ability of PUs, we can identify two types of PU. The first type refers to PUs containing only CPUs that are not equipped with any GPUs. The second type refers to PUs containing both CPUs and GPUs. The GPUs, equipped with many cores, offer massive computing capability as well as high parallelism. However the SIMD architecture of GPUs makes them very sensitive to thread divergence while the SPMD design of CPUs is less sensitive.

On the other side, as discussed in Section 1.3, two types of parallelism are mostly considered in general for the generic B&B algorithm. At the node-based parallelism, several B&B tree nodes can be bound *in parallel*. At the tree-based parallelism, several B&B subtrees can be explored *in parallel*. The CPUs deal with the tree-based parallelism in order to manage irregularity generated at runtime of parallel B&B. The GPUs are mostly suitable for the node-based parallelism so that the bounding operation is managed inside the GPUs while the other operations are performed by the GPUs' host (i.e CPUs). In fact, due to the irregularity and unpredictable shape of the tree, it is well understood that implementing the whole search operations inside GPUs, could suffer from the thread divergence induced by the SIMD programming model of GPUs. The bounding operation, on the other hand, can highly benefit from the parallelism offered by many GPU cores.

Although GPU devices can handle the evaluation of many tree B&B nodes *in parallel* [Chakroun 2012, Lalami 2012], the CPU hosts still have to prepare a data containing these nodes, copy them into GPU memory and copy back the results. This implies that while computations are carried out on GPU device, CPU host is idle and vice-versa. In our approach, the CPU host and the GPU device are

managed to run computations *in parallel*, i.e., while the GPU device is evaluating tree nodes, the CPU host is preparing new tree nodes for the upcoming evaluation in the GPU device.

3.2.2 Workload Irregularity (Q2)

It is essential to fully explore computing resources of a single CPU-GPU. However, it is more challenging to fully utilize the networked resources that is available in distributed computational environments. In fact, the irregularity generated at runtime can eventually lead to very poor performances because most computing nodes are underloaded and few others are highly overloaded, or because of the cost of synchronizing PUs and transferring work is so high. We decided to use Random Work Stealing (RWS) to tackle the irregularity problems among distributed PUs for node-heterogeneous platforms due to the following reasons. Firstly, RWS is theoretically proved to be an optimal solution to deal with irregularity issues under some constraints [Blumofe 1999]. Secondly, we observed that RWS produces good performance at small and average scales without any tuned parameters. Lastly, comparing to the tree-based approach presented in Chapter 2 or other overlay-based approaches, RWS is less sensitive to where PUs are placed in a logical graph as RWS can be seen as a clique and each vertex of the graph is equivalent.

3.2.3 PUs Compute Power (Q3)

The potential computational capability of different types of PUs might be different in order of magnitude. For instance, the relative performance of CPU and GPU can range from dozens to some hundreds orders in the favor of GPU. Since the bounding operation is the most time consuming in a B&B process and it can be implemented inside the GPU. Computing the bound for thousands of B&B nodes in parallel can result in an impressive acceleration. However, a GPU can only run with its maximum capacity when an accurate amount of B&B nodes is provided as input. Therefore there will be a significant lost if a GPU is provided less than its required value.

To understand the issues of node-heterogeneity of distributed PUs, one has to keep in mind that (i) a GPU is substantially faster in evaluating B&B tree nodes than a CPU, (ii) nothing can be assumed about the amount of tree nodes initially. Hence, if GPUs run out of work and stay idle searching for work, the performance of the system can drop dramatically. If only few CPUs are available in the system, work stealing operations from CPUs to GPUs can cause a severe penalty to performance. This is because the few CPUs can only contribute very little to the overall performance but their stealing operations to GPUs can disturb the GPU computations and prevent them from reaching their maximal speed. In contrast, if work is scheduled more on GPUs, then a significant loss in performance can occur when a relatively large number of CPUs are available.

To tackle these issues, we propose to estimate the compute capability of PUs

continuously at runtime with respect to the problem instance being executed so that tasks can be offloaded based on the normalized power of thieves and victims.

3.3 The *2MBB* architecture: *M*ulti-CPUs *M*ulti-GPUs Parallel *B&B*

In this section, we will describe in detail our approach in the scenarios of Multi-CPUs and Multi-GPUs. In these scenarios, all PUs are distributively connected and each PU is either a single CPU or a single CPU equipped with a GPU device. Fig. 3.1 presents the big picture as well as different components of our framework. We will detail them in the next subsections.

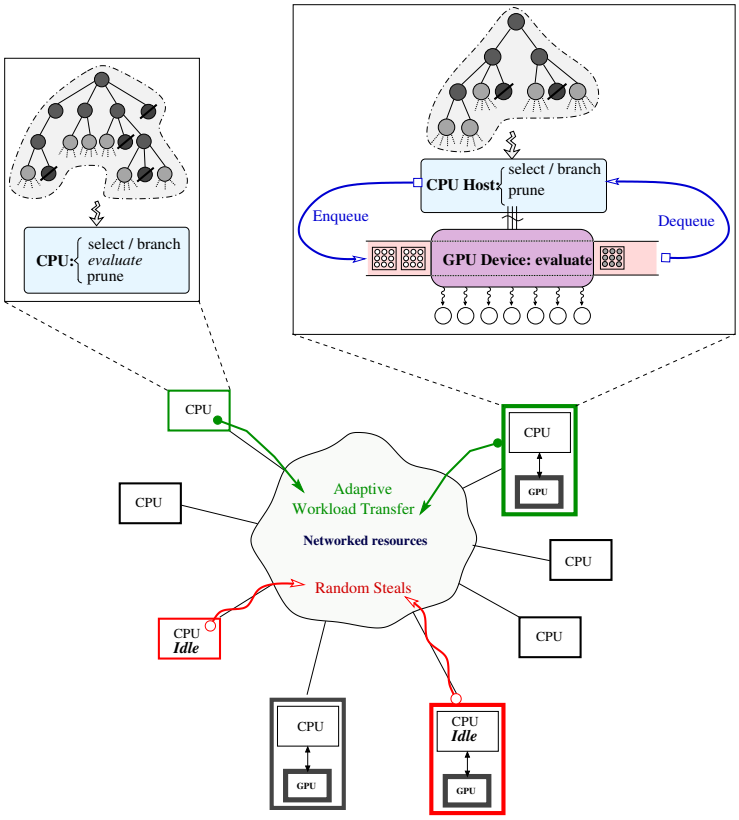

Figure 3.1: Overview of the 2MBB architecture: Multi-CPUs Multi-GPUs for B&B

3.3.1 Host-device parallelism for single CPU-GPU

For the PUs equipped with GPUs as depicted in Fig 3.1, the select, branch and prune operations are performed by the CPU host and the bound evaluation is handled by the GPU device. Generally speaking, for each bounding evaluation running on a GPU, input data comprising several B&B tree nodes is transferred to GPU

memory, a kernel is executed on the input and the outputs are copied back to the CPU host for being processed (i.e prune operations). In other words, standard CPU/host-GPU/device executions are synchronized sequentially. While the CPU host is performing select, branch or prune operations in order to prepare a new input data for GPU device, or even while copying data to/from device, the GPU is stalled. Similarly, while the evaluation of B&B tree nodes is running on the GPU device, the CPU host is stalled. This can significantly slow down computations especially when the CPU host and the GPU device can perform concurrent operations *in parallel*.

With the rapid evolving of GPU devices, it is now possible to address the above issue by carefully exploiting the new available hardware and software technologies. For instance, NVIDIA GPUs with compute capability ≥ 1.1 are associated with a *compute* engine and a *copy* engine (DMA engine). NVIDIA's Fermi GPUs have up to 2 copy engines, one for uploading from CPU host to GPU device and one for downloading from GPU device to CPU host. Each engine is equipped with a queue to store pending data and kernels that will be processed by the engine shortly.

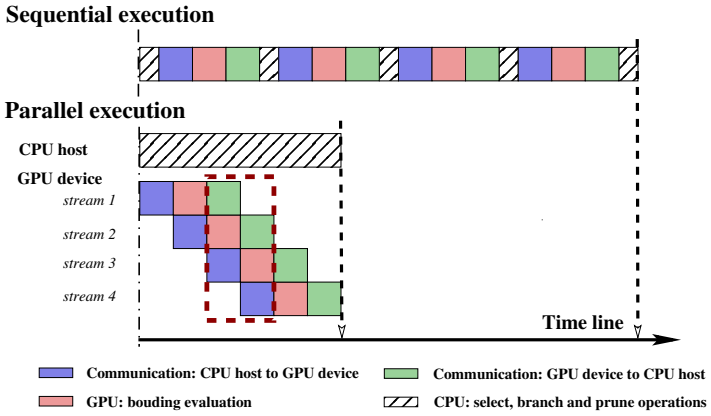

Figure 3.2: Example of host-device parallelism for single CPU-GPU

The host-device parallelism discussed in our approach can be enabled using CUDA primitives as sketched in Algorithm 6. Each ENQUEUE procedure dispatches CUDA operations into the GPU device *asynchronously*, i.e pushes/retrieves data and launches the kernel. This is possible by wrapping those operations into a CUDA stream. All operations inside the same CUDA stream get automatically synchronized and executed sequentially, but the CUDA operations of different streams could overlap one with the other, e.g., execute the kernel of stream 1 and retrieve data from stream 2 concurrently in parallel. In our implementation, we use a maximum number of streams, i.e., variable $r_{\max}$, which is the maximum number of elements *(data, kernel)* in the queue of GPU Copy engine and Compute engine. The maximum number of streams that a GPU can handle depends in general on GPU global memory characteristics. Asynchronously in parallel to the ENQUEUE procedure, the DEQUEUE procedure in Algorithm 6 waits for data copied back from the device on a given CUDA stream, and processes the output data. Figure 3.2 briefly illustrates

the host-device concurrent parallelism of four CUDA streams comparing with the sequential one.

Algorithm 6: CPU-GPU parallelism — Concurrent host-device template

Data: q_host, q_device: queue of *task* in host and GPU; q_host_size: current size of q_host (0 initially); $stream[r_{\max}]$: CUDA Stream of $r_{\max}$ elements; w_index, r_index: next index to write (resp. read) to (resp. from) the queues (0 initially).

```
1 while tree nodes are available do in parallel:
    // Push tree nodes for evaluation inside GPU
2   Execute Procedure ENQUEUE;
    // Retrieve and process evaluated nodes from the GPU
3   Execute Procedure DEQUEUE;
```

Procedure Enqueue

```
1 while q_host_size < r_max do
2   q_host[w_index].task ← prepare a pool of tree nodes;
    // Asynchronous Operations on stream[w_index]
3   cudaMemcpyAsyn(q_device[w_index], q_host[w_index], sizeof(q_host[w_index].task),
4                  cudaMemcpyHostToDevice, stream[w_index]);
    // Launch parallel evaluation (bounding) on device
5   KERNEL<<< stream[w_index] >>> (q_device[w_index]) ;
6   cudaMemcpyAsyn(q_host[w_index].bound, q_device[w_index].bound,
7                  sizeof(q_device[w_index].bound),
8                  cudaMemcpyDeviceToHost, stream[w_index]) ;
9   w_index ← (w_index + 1) (mod r_max);
    q_host_size ← q_host_size + 1 ;
```

Procedure Dequeue

```
1 if q_host_size > 0 then
    // Wait for results from device on stream[r_index]
2   cudaStreamSynchronize(stream[r_index]) ;
3   Process output data from q_host[r_index], i.e., prune nodes ;
4   r_index ← (r_index + 1) (mod r_max) ;
5   q_host_size ← q_host_size − 1 ;
```

3.3.2 Adaptive Stealing for Multi-CPUs Multi-GPUs

We decided to use Random Work Stealing (RWS) to balance the workload among PUs. In this context, the stealing granularity, that is the amount of B&B tree nodes

to be offloaded from victims to thieves for stealing operations, denoted f, plays a crucial role. Depending on the hardware platform and the input application, there may exist a value of stealing granularity giving the best performance. In general, a thief attempts to balance workload evenly between itself and the victim. In fact, when this amount of work is very small, the large overhead is observed since many load balancing operations are performed. At the opposite, when it is very large, too few load balancing operations will occur, thereby resulting in large idle times despite the fact that surplus work could be available. In classical RWS approaches, this is a hand-tuned parameter which depends on the distributed system and the application context [Min 2011]. In a theoretical study [Blumofe 1999], the stability and optimality of RWS can be analytically guaranteed for $f \leq 1/2$. In practice, the so called steal-half strategy ($f = 1/2$) is often shown to perform efficiently using homogenous computing units. Besides, in a heterogeneous and hybrid computing system, the hardware characteristics of PUs, e.g., clock speed, Cache, RAM, etc, can be highly needed to balance the work load evenly depending on the characteristics of every available PU. Because high variations in computing power among PUs can lead to high imbalance and idle times, one has also to manage this issue carefully when distributing work. One possible solution for the above issues could be to profile the system components/PUs and tune work granularity offline before application execution in order to get the best performance. It should be clear that such an approach is not reasonable nor feasible, for instance when the system may undergo a huge number of many different types of PUs, or when having many different applications at hand.

In our stealing approach, we make every PU maintain at runtime a measure reflecting its computing power, i.e., variable X in Algorithm 7. As the computations are running on, every PU adjusts its measure continuously with respect to the work processed in the previous iterations. In our approach, we simply use the average time needed for processing a B&B subproblem. More precisely, each PU sets its computing power to be $X = N/T$, where T is the (normalized) computing time elapsed since the PU has started the computation and N is the number of tree nodes explored locally by that PU. Notice that time T includes, in addition to tree node evaluation (i.e. B&B lower bounding), the time needed for other search operations (i.e. select, branch and prune) but *not* the time when a PU stays idle. When running out of work, a PU v then attempts to steal work by sending a request message to another PU u chosen at random, while wrapping the value of X in the request. If a victim has some work to serve, then the amount of work (i.e., number of tree nodes) to be transferred is in the proportion of $X/(X+Y)$, where Y is the computing power maintained locally by the victim. Otherwise, a reject message is sent back to notify the thief and a new stealing round is performed. Initially, the value of X is normalized so that all PUs have the same computing ability. In other words, the system starts stealing half and then the stealing granularity is refined for each pairwise PU. Intuitively, each PU acts as a black-hole, so that the higher computing power of PUs is, the more available work are flowed to the black-hole. Furthermore, no knowledge about PUs is needed so that any performance variation

at system/application level would also be detected at runtime.

Algorithm 7: Random Work Stealing for Multi CPUs Multi GPUs

```
1 while termination not detected do
2     if local queue is empty then
3         Execute Procedure THIEF;
4     else
5         Execute Procedure VICTIM;
```

Procedure Thief

```
1 X ← runtime normalized computing power ;
2 repeat
3     u ← VICTIM_SELECTION_AT_RANDOM;
4     SEND a steal request message with X to u;
5     RECEIVE u's response (reject or work) message ;
6     if a steal request is pending then
7         v ← pull the next pending thief request;
8         Send back a reject message to v ;
9 until retrieving work from victim u or termination detected;
```

Procedure Victim

```
1  if a steal request is pending then
2      if tasks are available then
3          Y ← runtime normalized computing power ;
4          (v, X) ← pull the next pending thief request;
5          work ← SHARE_WORK in the proportion of X/(X+Y);
6          Send back shared work to v ;
7      else
8          Send back a reject message to v ;
9  else
10     task ← pop the next task from local pool;
11     EXECUTE task ;
```

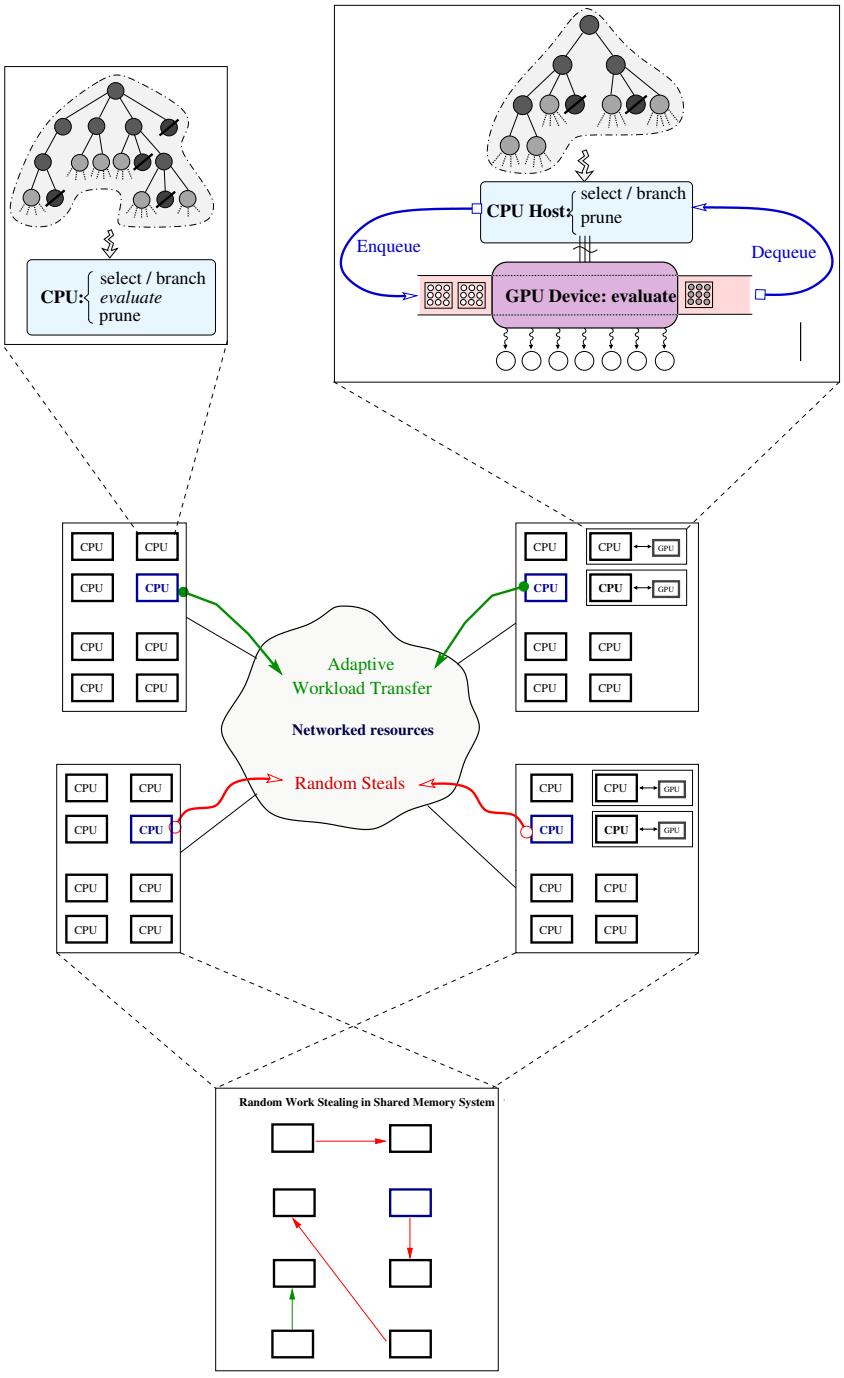

Figure 3.3: Overview of 3MBB architecture: Multi-cores Multi-CPUs Multi-GPUs for B&B

3.4 The *3MBB* architecture: *M*ulti-cores *M*ulti-CPUs *M*ulti-GPUs Parallel *B&B*

The previous section introduced our approach for balancing workload of parallel B&B in the scenarios of Multi-CPUs Multi-GPUs. However, many node-heterogeneous platforms come with multi-core processors, hence they have a mix of shared memory and distributed memory, which brings more challenges to the parallel computation as well as dynamic load balancing. In this context, there exists two levels of parallelism from the perspective of hardware-aware topology. *Intra-node* parallelism refers to parallel computations among cores of a single compute node through shared memory, while *inter-node* parallelism indicates parallel computations among compute nodes in distributed memory systems. The difference in communication cost between shared memory and distributed memory is very pronounced so that stealing tasks between cores in a single compute node is much faster compared with the one in distributed memory. Our approach takes the hardware topology into account and the algorithm tries to steal work from the lowest hierarchy (i.e intra-node) to the highest hierarchy (i.e inter-node).

Fig 3.3 describes the big picture as well as different components of our framework. In the considered platforms, each computing node is a multicore system, and some of them are equipped with GPU devices. Each GPU device is hosted by a CPU core. In our approach, we applied work stealing at two levels. In the first level, work stealing is performed in the multicore shared memory system among CPU cores. In the second one, work stealing is performed through distributed memory system as the 2MBB presented in the previous section. We will detail our approach in the next subsections. Let us remind that this nature design of work stealing has several advantages compared to the typical design of using a single global pool shared by many worker threads that are recently used in many works [Mezmaz 2013, Chakroun 2013b].

3.4.1 Intra-node parallelism

In a shared memory system, the communication cost among threads is very negligible which allows efficient parallel computations of B&B. We decided to use asynchronous multiple work pools so that each thread runs on a single physical core and manages a separate work pool storing new generated B&B subproblems. We adopted work stealing for multi-core systems such that a thread plays a role of either a thief or a victim. Whenever a thread finishes all subproblems of its work pool, it will become a thief and tries to steal works from other threads running on the same compute node. Let us remind that stealing operations and task offloading among cores are converted to read/write operations to/from common data structures therefore they are very fast compared with message-passing mechanism in distributed memory systems. However, simultaneous read/write operations to common data causes data race problems leading to unexpected outputs eventually. Locking and synchronizing simultaneous accesses of several threads are the popular solution in this context but

they come with a price. Overusing these techniques highly reduces the potential parallelism of a multi-core system. Therefore, achieving a good performance requires a well design of data structure in order to minimize locking operations as many as possible.

A naive approach is that each thread manages a fully shared work pool. In other words, a thread can fully access to the work pools of other threads during execution. This approach is simple in design as well as implementation but it introduces high overhead as locking techniques are used to synchronize multiple accesses to a work pool of a thread. For instance, when a thief thread tries to steal work from another victim thread, it has to lock the work pool of the victim then offloads tasks from the victim's work pool to the thief's work pool. During the locking time, the victim is forced to be stalled as it can not access to its own work pool to pop/push task. Notice that a thread has also to perform a lock operation to its own pool whenever it accesses to the pool since the pool is fully shared with other threads.

A better approach is to split work pool into two different pools: one for the owner thread storing new generated tasks, one for storing sharable tasks for being stolen by other threads. Therefore, locking is avoided whenever the owner thread tries to access to its own pool, thus improving performance. However, tasks are copied backward and forward between the two pools whenever one of them is empty. The copy operations expose a shortcoming of this approach.

The most appropriate approach is to use a single work pool split into a private and a public part as described in [Dinan 2009]. The private region functions like the work pool which is exclusively owned by the owner thread. The global region is exposed to other threads. In a single work pool, these two regions are separated by a *split* pointer. Furthermore, the amount of tasks of the private and public regions are adjusted by moving the *split* pointer forward or backward without any memory copies.

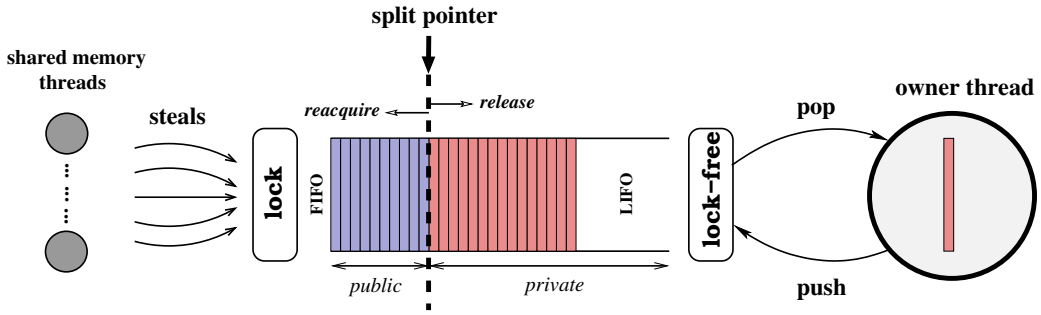

Figure 3.4: Split work pool

Figure 3.4 describes the design of split work pool. In detail, the private portion functions like a Stack with a LIFO manner and the public portion works like a Queue with a FIFO manner. The LIFO characteristics of the private region allows threads to perform a DFS search on a B&B tree in order to reach to a leaf quickly. The

FIFO property of the public region allows to share coarse grain B&B subproblems that potentially generate more descent subproblems. In B&B, subproblems that are close to the root usually contain large amount of works while this amount decreases as the recursion develops. Besides, steals of other threads to the public regions are handled by a lock, but the owner threads is lock-free with respect to its private region since it can freely push/pop task to/from it. A split pointer is associated with two operations: *release* and *reacquire*. The *release* operation refers to move the *split* pointer to the private region for exposing private tasks to public region. The *reacquire* operation indicates to move the *split* pointer to the public region for moving public tasks to the private region. In our design, we define two types of *split* pointer:

- **CPU-based split pointer**: refers to performing the *reacquire* operation whenever the private region is empty. As CPU threads can only process a single task at once. We also call it as *strictly split pointer*.
- **GPU-based split pointer**: refers to perform the *reacquire* operation in advance before the private region is completely empty. As GPU threads can handle several tasks at once, the maximum potential compute power of GPUs is only guaranteed if and only if they are provided large enough tasks as inputs. In this case, if the current tasks of the private region does not fit as a good input for GPUs, the *reacquire* operation is performed to increase the input size for GPUs.

Termination. the intra-node approach can be borrowed to easily handle termination by making available threads handle a same shared variable. In fact, the value of the termination variable refers to how many idle threads exist in the system. Therefore the variable is increased by one if a thread becomes idle, and it is decreased by one if an idle thread becomes busy. When the value is equal to the number of running threads, termination can be detected locally within a shared memory component. Locking techniques are used when updating the termination variable. This approach does not scale due to the central nature, but it's still acceptable in shared memory systems.

Knowledge Sharing. Each thread manages a separate data structure for storing the best solution found so far. Whenever a thread finds a better solution, it will simply update the value to all other threads. Locking technique is also used when updating the best-found solution.

3.4.2 Inter-node parallelism

So far we have presented the design of intra-node parallelism to parallelize and to balance workload of B&B applications in shared memory systems. To expand the intra-node parallelism on distributed memory systems, stealing operations must be transformed to messages that are sent across compute nodes through the underlying interconnect networks. One naive approach is to allow individual threads perform

random work stealing as the 2MBB presented above with one main difference. Stealing operations are performed differently according to the relative location of thieves and victims. In more details, when a thief and a victim thread share the same compute node, the stealing operation is performed as the intra-node parallelism presented above. Otherwise, the thief sends a stealing message to the victim. This approach is straightforward but exposes some shortcomings. In fact, thieves might suffer from high communication overhead since stealing across compute nodes is much more expensive than stealing in the shared memory system. Instead, they must take into account the usage of the multicore systems in priority compared with the distributed systems.

In our approach, stealing across compute nodes is only enabled when all threads detect that there is no work available locally. Here, a single compute node is meant as a shared memory system where several compute cores are available. This allows all threads of a compute node try to complete all available local works before stealing works from other distributed compute nodes. However, if all idle threads try to steal works through the distributed memory system, they might generate large amount of messages and cause unnecessary overhead for processing them. This led us to design an approach where stealing across compute nodes is only handled by a single thread in each distributed node. In fact, this thread is responsible for processing and handling all incoming messages of the other nodes and it plays the role as a leader or a master thread in the corresponding compute node.

The master thread functions as the other threads with one main difference. When it detects that there is no work in the work pools of itself and the other threads, it will perform the random work stealing work across compute nodes. The master thread randomly selects another compute node as a victim and sends a steal request message. In our design, we use the adaptive work sharing policy in order to evenly share works among compute nodes, but with two main differences as the following. The first one is that the power of a compute node is measured as an aggregated value of all the threads of a node. The second one is that the amount of work to be considered at victims is the amount of all available tasks in the public regions of all local threads. In more details, the master thread of a thief i collects the aggregated computing power $X = \sum_j x_{i,j}$ (where $x_{i,j}$ is the computing power of thread j at thief i) and sends message to a randomly selected victim p while wrapping the value of X. Similarly, upon receiving a steal message, the master thread of the victim p also measures its aggregated computing power $Y = \sum_q y_{p,q}$ (where $y_{p,q}$ is the computer power of thread q at the victim p), and then the amount of work to be transferred is in the proportion of $\frac{X}{X+Y}$. Technically speaking, the master thread of the victim p collects $s_{p,q} = t_{p,q} \cdot \frac{X}{X+Y}$ work units from the public region of every work pool of all the local threads (where $t_{p.q}$ is the total available works of the public region of thread q at the victim p), and shares the total amount of $S = \sum s_{p,q}$ work units to the requesting thief. Figure 3.5 details all the working states of this approach.

Knowledge sharing and Termination. Both issues are managed using a

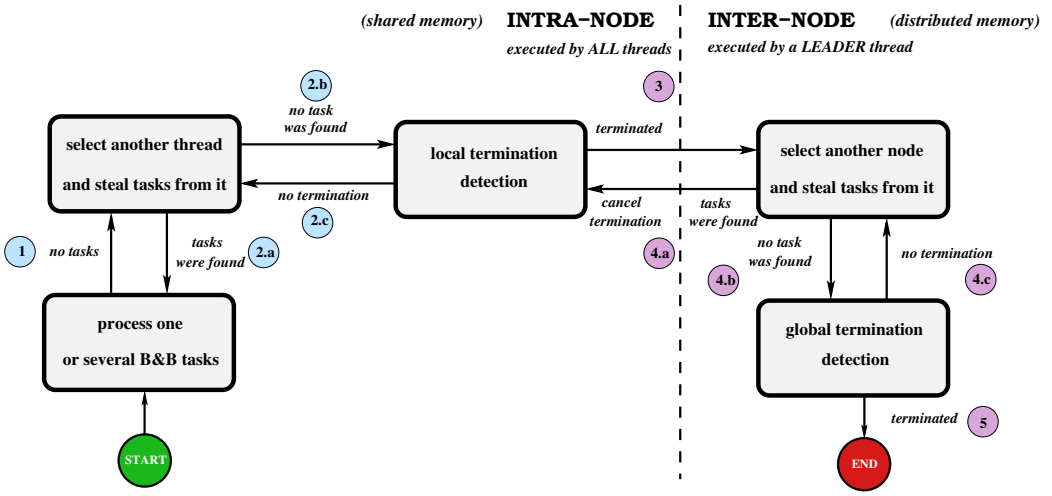

Figure 3.5: Overview of 3MBB architecture: Multi-cores Multi-CPUs Multi-GPUs for B&B

binary tree as for the random work stealing approach presented in Section 2.3. Notice however that the tree is only used to span distributed compute nodes (that is the master threads) and not all available cores.

3.5 Experiments

3.5.1 Experimental Setting

Three clusters C_1, C_2 and C_3 of the Grid'5000 French national platform [Grid'5000] were involved in our experiments. Cluster C_1 contains 10 nodes, each equipped with 2 CPUs of 2.26Ghz Intel Xeon processors with 4 cores per CPU. Besides, each node is coupled with two Tesla T10 GPUs. Each GPU contains 240 CUDA cores, a 4GB global memory, a 16.38 KB shared memory and a warp size of 32 threads. Cluster C_2 (resp. C_3) is equipped with 72 nodes (resp. 34 nodes), each one equipped with 2 CPUs of 2.27 Ghz Intel Xeon processor with 4 cores per CPU (resp. 2 CPUs of 2.5 Ghz Intel Xeon processor having 4 cores) and a network card Infiniband-40G.

We use the standard Taillard's Flowshop instances in the family 20∗20. Only the bounding operation has to be executed inside the GPU devices. We use the kernel implementation provided by [Chakroun 2012, Melab 2012] and use it as a black-box. Let us point out that the GPU *kernel* implementation of [Chakroun 2012, Melab 2012] has a parameter s referring to the maximum number of B&B tree nodes that are pushed into GPU memory for parallel evaluation. It is shown in [Melab 2012] that the parameter s has to be fixed to a value $s^\star$ so that the device memory is optimized and the performance is the best on a single GPU. Since we assume that the GPU kernel is provided as a black-box, and unless stated explicitly, the value of s is fixed in our experiments to be simply $s^\star$. In our experimental

study, we are also interested in analyzing how our approach would perform when having GPU kernels allowing for different speed-ups in the evaluation phase. This can be typically the case for other type of problems, different hardware configurations, etc. Being able to understand whether our load balancing mechanism is efficient in such a heterogeneous setting, independently of the considered scale or speedup gap between available CPUs and GPUs, is of great importance. In this chapter, we additionally view the parameter s as allowing us to empirically reduce the intrinsic speed of a single GPU, and thus to experiment our approach while using different GPU and CPU configurations.

In the remainder, we shall evaluate our 2MBB and 3MBB approach in several scenarios. Two sets of experiments are designed as following:

- **We consider the following set of experiments for 2MBB.**.

  1. Running our approach with a single CPU-GPU.
  2. Running our approach at different scales with Multi GPUs.
  3. Running our approach with a fixed number of GPUs, while scaling the CPUs.
  4. Running our approach with a fixed number of CPUs, while scaling the GPUs.
  5. Running our approach with CPUs and GPUs having different computational powers.

- **We consider the following set of experiments for 3MBB and 2MBB**.

  6. Comparing our approaches when running them with multi-core CPUs and multiple GPUs in large scales.
  7. Comparing our approaches when running them with multi-core CPUs in large scales (without GPUs).

Regarding the experiments of 2MBB, each processing unit is launched on one CPU core or on a CPU core equipped with a GPU device (taken from C_1). For the first four scenarios, CPUs are taken from cluster C_2. As for the fifth scenario, we mix CPUs of different hardware clock speeds, taken from C_2 and C_3, and GPUs launching kernels configured with different values of s. The previous scenarios aim at providing insights on how the system performs independently of the scale and/or the power of CPUs and GPUs. Regarding the experiments of 3MBB, the CPUs are taken from clusters C_1, C_2, $C3$ and the GPUs are taken from C_1. For all experiments, we measure T and N, respectively the time needed to complete the B&B tree search and the number of B&B tree nodes that were effectively explored. All reported speedups are relative to the number of B&B tree nodes explored by time units, that is N/T.

3.5.2 Performance of 2MBB architecture

3.5.2.1 Impact of host-device parallelism in single CPU-GPU

We start our analysis by evaluating the impact of host device concurrent operations. For the ten instances in Taillard's family 20*20, we report in Fig 3.6 execution time and speedup w.r.t. the baseline sequential host-device execution [Melab 2012], for different number of concurrent CUDA streams (variable $r_{\max}$ in Algorithm 6) and different GPU kernel parameters s. One can clearly see that substantial improvements are obtained, i.e., our approach is at least twice faster. It also appears that the maximum number of concurrent CUDA streams $r_{\max}$, which is the only parameter used in our approach, has only a marginal impact on performance. Fig 3.6 Right shows that the speed-up, w.r.t the sequential host-device execution, is substantial ($> 30\%$) but depends on kernel parameter s. This is because for lower values of s, the host spends more time pushing small amount of data, while the device is less efficient. In other words, host-device parallelism performs better when the amount of data and computations on device is higher.

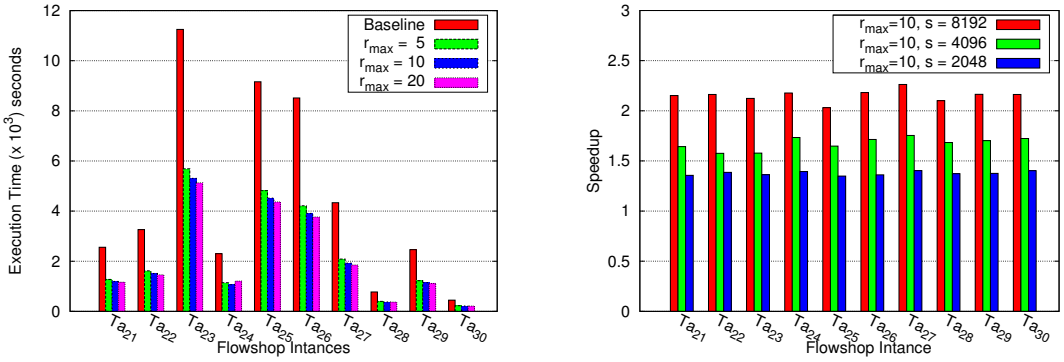

Figure 3.6: Host-device parallelism *vs.* baseline sequential host-device execution [Chakroun 2012]. **Left**: Execution time with different number $r_{\max}$ of CUDA streams and $s = s^{\star}$ (Lower is better). **Right**: Speedup w.r.t baseline for different values of s and $r_{\max} = 10$ (Higher is better).

3.5.2.2 Scalability and Stealing Granularity for Multi GPUs

In this section, we study the scalability of our approach when only multiple GPUs are available in the system. For this set of experiments we choose the first instance Ta_{21} to be our case study. In Fig 3.7 Left, we report the speedup of our approach w.r.t one single GPU, and also the speedup obtained when using a static stealing granularity (with of course host-device parallelism enabled). By static stealing, we mean that we initially fix the proportion of tree nodes to be stolen as a parameter $f \in \{1/2, 1/4, 1/8\}$. Two observations can be made. Firstly, our adaptive approach performs similar to the best static stealing, which is for $f = 1/2$ from our experi-

ments. Other values of f in static stealing are in fact worse especially in high scales. Secondly, we are able to scale linearly with the number of GPUs. At scale 16, one can notice a slight decrease in speedup. We attribute this to two factors: (i) the communication cost of distributing work strategy to be not negligible in large scales, and (ii) sharable work becomes very fine grain so that it limits the maximal performance of GPUs. Actually, the results of Fig. 3.7 Left are obtained with parameter s being $s^\star$ the maximal (and best) amount of tree nodes that a single GPU can handle. In Fig 3.7 Right, we push our experiments further by taking other values for parameter s. We can clearly see that the speed-up (w.r.t. one single GPU running a kernel with the same value of s) is not impacted. The scalability is even slightly better when the kernels are less efficient. This can be interpreted as the scalability of our approach being not sensitive to other system/application settings with GPUs having possibly different processing powers.

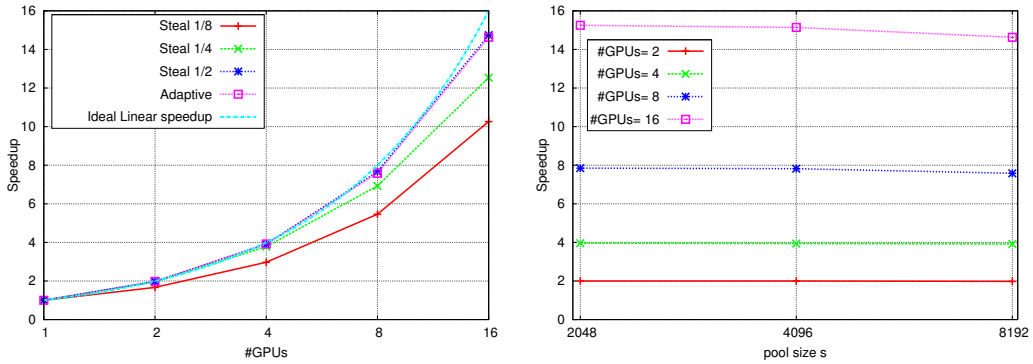

Figure 3.7: **Left**: Scalability of our adaptive approach *vs.* static stealing ($s = s^\star$). X-axis refers to the number of GPUs in log scale. Y-axis refers to the speed-up with respect to one GPU. **Right**: Speedups of our approach as a function of s. $r_{\max} = 10$.

3.5.2.3 Adaptive Stealing for Multi-CPUs Multi-GPUs

In this section, we study the properties of our approach when mixing both CPUs and GPUs. For that purpose, we proceed as following. Let α_i^j be the speedup obtained by a *single* PU j with respect to PU i. We naturally define the linear (ideal) normalized speedup *with respect to PU* i, to be $\sum_j \alpha_i^j$. For instance, having p *identical* GPUs and q *identical* CPUs, each GPU being β times faster than each CPU, our definition gives a linear speedup with respect to *one GPU* (resp. *one CPU*) of $p + q/\beta$ (resp. $q + \beta \cdot p$). The following sets of experiments shall allow us to appreciate the performance of our approach when varying substantially the ratio between the number of GPUs and CPUs.

Mixed Scaling. Our first set of experiments is complex since we manage to mix multiple GPUs with empirically different powers and multiple CPUs with different

clock speeds. This scenario is in fact intended to reproduce a heterogeneous setting where, even PUs in the same family do not have the same computing abilities. In this kind of scenario, where in addition the power of PUs can evolve, e.g., due to system maintenance constraints or hardware renewals/updates, even a weighted hand tuned steal strategy is not plausible nor applicable. In the results of Fig. 3.8, we fix the number of CPUs to 128 with half of them taken from cluster C_2 and the other half from cluster C_3 (C_2 and C_3 have different CPU clock speeds as specified previously). For GPUs, we proceed as following. We use a variable number of GPUs in the range $p \in \{1, 4, 8, 12, 16, 20\}$. For $p > 1$, we configure the system so that 1/2 of GPUs run a kernel with pool size $s^\star$, 1/4 of them with pool size $s^\star/2$ and the last 1/4 of them with pool size $s^\star/4$. Once again our approach is able to adapt the load for this complex heterogeneous scenario and to obtain a nearly optimal speedup while outperforming the standard steal-half strategy. From the previous set of experiments we can thus conclude that our approach allows us to take full advantage of both GPU and CPU power independently of considered scales, or any hand tuned parameter.

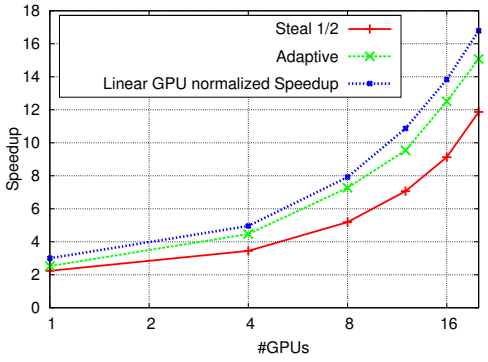

Figure 3.8: Speedup when scaling heterogenous GPUs (1/2 with $s^\star$, 1/4 with $s^\star/2$, 1/4 with $s^\star/4$), and 128 heterogenous CPUs (1/2 from cluster C_2, 1/2 from cluster C_3). Speedup is w.r.t. one GPU configured with $s^\star$. $r_{\max} = 10$.

GPU Scaling. We now fix the number of CPUs and study how the behavior of the system when scaling the number of GPUs. Results with 128 (identical) CPUs and (identical) GPUs ranging from 1 to 16 are reported in Fig 3.9. We can similarly see that our adaptive approach is still scaling in a linear manner while being near optimal. It is also substantially outperforming the static steal-half strategy.

CPU Scaling. In this set of experiments, we fix the number of GPUs and scale the number of CPUs. Besides, we experiment two other static baseline strategies. The first one is the standard steal-half strategy. The second one, we term 'Weighted Steal', is hand tuned as following. After profiling the different PUs in the system and running the B&B tree search with the corresponding FlowShop instance on *every single PU until termination*, we provide each PU with the relative computing power of every other PU in the system. Then, the amount of work transferred from

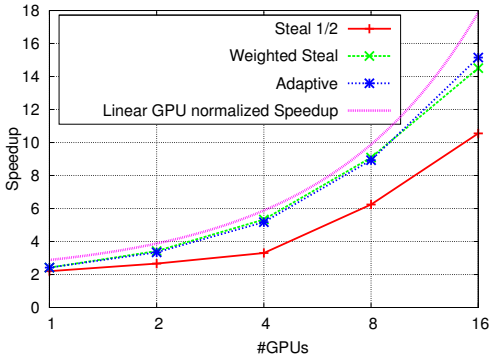

Figure 3.9: Speedup (w.r.t. one GPU) when scaling GPUs and using 128 CPUs. $r_{\max} = 10$.

PU i to PU j is initially fixed to be in the proportion of the relative computing power observed in the profiling phase. The results with 1 and 2 (identical) GPUs and (identical) CPUs ranging from 1 to 128 are reported in Fig 3.10.

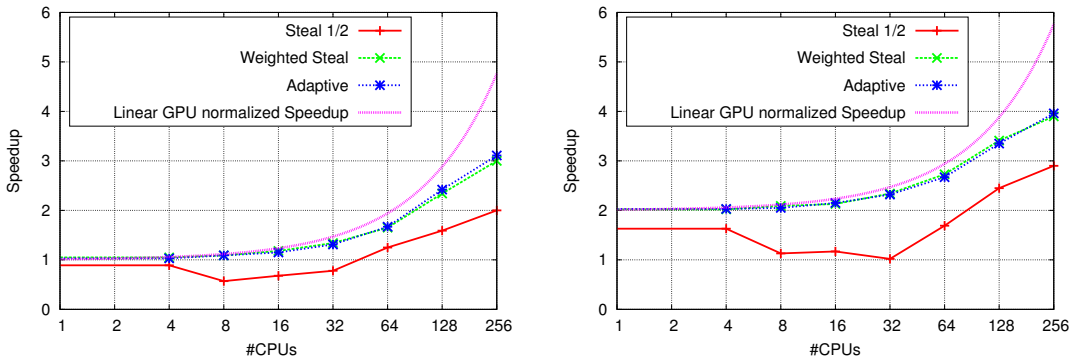

Figure 3.10: Speedup of our approach *vs.* static steal when scaling CPUs and using 1 GPU (**Left**) and 2 GPUs (**Right**). X-axis is in the log scale. Speed-up are w.r.t. one GPU. $r_{\max} = 10$.

One can clearly see that our approach performs similar to the weighted static strategy while avoiding any tedious profiling and/or PU code configurations. In particular, the weighted strategy cannot be reasonable in production systems with different PU configurations since it requires much time to tune the systems. Turning to the steal half static strategy, it appears to perform substantially worse. When having relatively few CPUs, the performance of steal half is even worse than in a scenario where only GPUs are available (see Fig. 3.7). It is also getting worse as we push additional few CPUs in the system. Improvements over 1 or 2 GPUs are only observed when the number of CPUs is relatively high (w.r.t GPU power). Besides, our approach is able to scale near linearly at small and immediate scales up to

128 CPU cores. However, its performance significantly drops at larger scales (256 and 512 CPU cores) compared with the linear speedup. At the large scales, work become more fine-grained so that PUs spend more time in fetching them compared with the small and intermediate scales. In the next subsection, we will present the performance of 3MBB architecture while running at the large scales.

3.5.3 Performance of 3MBB architecture

In this section, we analyze the performance of our approach when dealing with multi-core platforms. In order to push our experiment further, we consider to solve the Taillard's instances at a larger scale of 512 CPU cores. Let us remind that in large scale experiments the application granularity might be broken into fine-grained which pushes more challenges on workload balancing techniques. In fact, the fine-grain aspects of the irregular applications can make PUs look for work more often, thus reducing the utilization of PUs in computation. The objective of the experimental study presented in this section is to evaluate the design of multi-core approach of 3MBB for B&B.

3.5.3.1 Relative scalability of 3MBB and 2MBB

In Fig 3.11, we analyze the scalability of 3MBB compared to 2MBB in different heterogeneous scenarios where the number of GPUs is fixed in the range 0, 1, 2, 4 and the CPU-cores are scaled from 1 up to 512. The results clearly show that the 3MBB approach significantly improves the performance of 2MBB. More precisely, the performance of the two approaches is equivalent in small and average scales (up to 128 CPU cores) where they both scale near linearly. In small and average scales, B&B workload is relatively not very fine-grained, hence decreasing the need for PUs to steal work units more frequently. This results in increasing the utilization of PUs in computations and consequently in better speedups. However, the performance of 2MBB starts to drop in the scale of 256 and 512 CPU-cores and it is substantially outperformed by the 3MBB approach. This is clearly attributed to the advanced and hybrid load balancing mechanism used in 3MBB which allows us to take full advantage of the different levels of parallelism exposed by the compute environment. In fact, balancing B&B workload locally at every shared memory component allows PUs to significantly reduce the idle times and the cost of stealing using message passing. We shall analyze this aspect in more details in the next subsection.

3.5.3.2 Large scale analysis

To appreciate the gain we can obtain for parallel B&B when using hybrid load balancing with both shared memory and message passing, we consider to solve the ten Flowshop instances in the scenario where no GPUs are used but only multiple CPUs with multi-core systems at the largest scale of 512 cores. The results reported in Fig. 7 clearly show that the 3MBB significantly improves the 2MBB by 30% (for instance Ta23) up to 50% (for instance Ta30). Notice that according to Fig. 3.12,

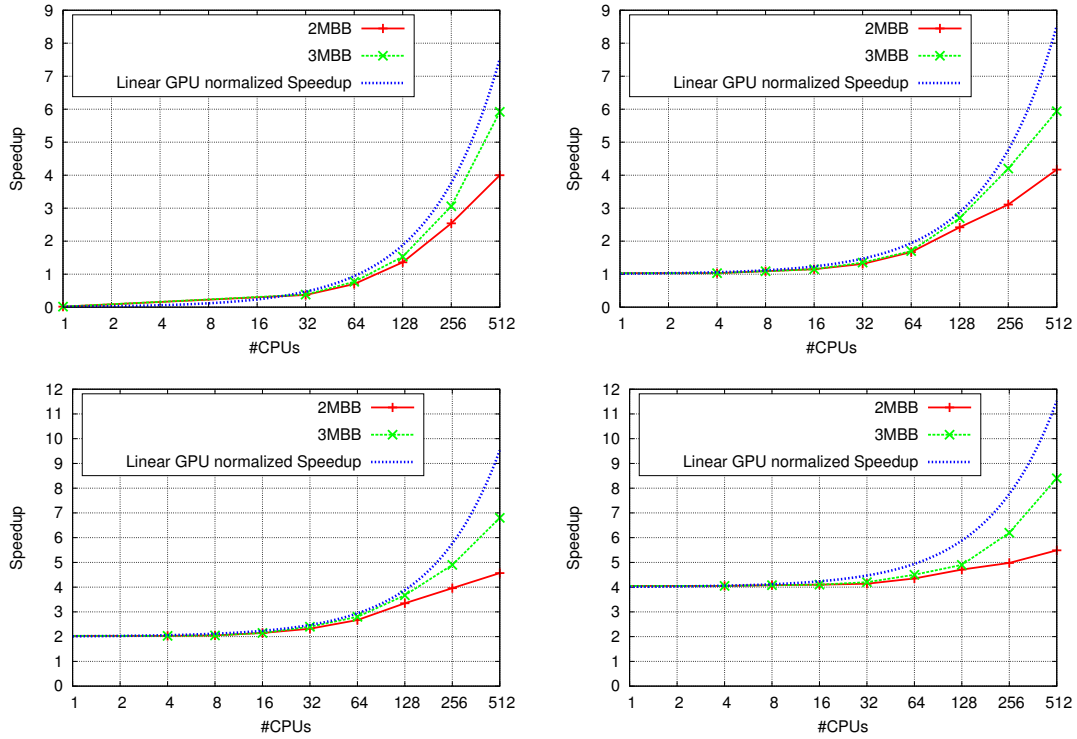

Figure 3.11: Speedup of 3MBB *vs.* 2MBB when scaling CPUs and using 0 GPU (**Top Left**), 1 GPU (**Top Right**), 2 GPUs (**Bottom Left**) and 4 GPUs (**Bottom Right**). X-axis is in the log scale. Speed-up are w.r.t. one GPU. $r_{\max} = 10$.

instance Ta30 (resp. Ta23) is likely to expose the most (resp. least) fine-grained parallelism.

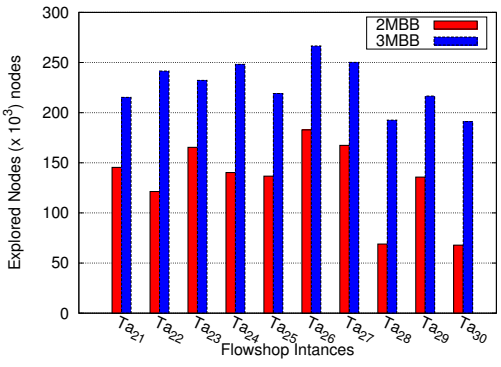

Figure 3.12: Explored Nodes of 3MBB *vs.* 2MBB on 512 cores

Table 3.1 allows us to get more insights on the impact of the shared memory parallelism incorporated in the 3MBB approach. More precisely, we measure and

report therein different statistics concerning the time where a PU stays idle searching work. Firstly, we can see (last column of Table 3.1) that the proportion of time that a PU spends requesting work over all B&B execution is significantly smaller for 3MBB (6.79% in average and over all instances) compared to 2MBB (32.88%). This indicates that the 3MBB allows us to maximize useful B&B computations over work transfer. This also shows that balancing B&B work efficiently among PUs is the critical issue when considering fine-grain large scale scenarios. Secondly, we can see that much less inter-node communications are performed by 3MBB compared to 2MBB (by a factor of 40.5 times in average). In fact, only the leader threads are allowed to perform inter-node steals in contrast to 2MBB where all individual threads execute inter-node steals. Not only the number of inter-node steals is significantly smaller for 3MBB, but they are also much less time consuming (by a factor of 3.43 times in average). This is attributed to the fact that a PU is likely to receive more work requests simultaneously, thus inducing more delays for responding them. Secondly, we can see that the intra-node steals are dominant compared with the inter-node ones which illustrates how the 3MBB approach prioritizes intra-node steals over inter-node steals. Since the cost of intra-node communication is relatively low and the design of the split work pool allows us to minimize the cost of thread synchronizations, the shared-memory computations are then optimized resulting in an efficiency mechanism for balancing the B&B tasks among the shared memory CPU-cores. On the other side, inter-node steals are much slower than intra-node steals; but they are very important to balance work load among distributed compute nodes. Let us remind that 3MBB tries to balance workload adaptively based on the aggregated power of available cores. A leader thread in 3MBB always tries to fetch enough tasks in order to serve all the threads sharing the same memory. Since only one single leader performs remote steals on behalf of many others, the scale of the distributed system is then artificially reduced to relatively few leaders and the inter-node communication is likely to be much more efficient.

The previous observations also hold when considering a large scale scenario incorporating GPUs devices as reported in Table 3.2 for the first Flowshop instance Ta21. We can see that 3MBB is not sensitive to the number of GPUs devices used. Furthermore, the reported results also confirm that this approach is able to reduce the overall communication cost, consequently limiting the penalty in performance one must pay when balancing B&B irregular workload distributively. However, notice from Fig. 3.11 that there is still a gap between the actual speedup of 3MBB and the linear one at the largest scale. We attribute this to the combined effect of two facts: (i) a single GPU device can only run at its maximum capacity only when there is enough B&B subproblems to push in, and (ii) the available B&B work gets more and more fine-grained and scattered over PUs at such large scales. Hence, although the GPU devices do not stay idle, it is more likely that they cannot acquire enough B&B subproblems to bound in parallel, hence decreasing the compute capacity of GPUs. In other words, as work is too fine-grained and the number of B&B subproblems that are pushed inside the GPUs is not optimal so that the GPUs are not able to reach their maximum speedups for such scales. Overall, we argue

		Inter-node			Intra-node			% of steal time
		# Steals	Time (s)	Time/Steal (s)	# Steals	Time (s)	Time/Steal (s)	
Ta_{21}	2MBB	1831	115.18	0.063		N/A		26.57%
	3MBB	226	3.59	0.016	3351813	5.69	0.0000017	3.96%
Ta_{22}	2MBB	1926	154.36	0.080		N/A		38.01%
	3MBB	193	2.43	0.013	3219090	5.47	0.0000017	2.78%
Ta_{23}	2MBB	3897	263.14	0.067		N/A		22.56%
	3MBB	153	3.02	0.019	3342713	5.68	0.0000017	0.89%
Ta_{24}	2MBB	1336	98.83	0.074		N/A		32.24%
	3MBB	145	6.60	0.045	2939638	5	0.0000017	5.47%
Ta_{25}	2MBB	2921	208.20	0.071		N/A		30.96%
	3MBB	195	4.13	0.021	4519199	7.23	0.0000016	2.23%
Ta_{26}	2MBB	3387	221.17	0.065		N/A		24.95%
	3MBB	168	2.90	0.017	4053746	6.89	0.0000017	2.20%
Ta_{27}	2MBB	3002	187.90	0.062		N/A		28.32%
	3MBB	177	2.62	0.015	3833216	6.516	0.0000017	3.26%
Ta_{28}	2MBB	830	76.54	0.090		N/A		47.93%
	3MBB	177	5.62	0.032	3053185	5.49	0.0000018	15.64%
Ta_{29}	2MBB	1323	95.38	0.072		N/A		28.35%
	3MBB	148	2.88	0.019	2649031	4.77	0.0000018	10.92%
Ta_{30}	2MBB	563	58.35	0.103		N/A		48.94%
	3MBB	127	2.74	0.021	2023738	3.64	0.0000018	20.58%

Table 3.1: 2MBB: time taken for inter-node steals

		Inter-node			Intra-node			% of steal time
		# Steals	Time (s)	Time/Steal (s)	# Steals	Time (s)	Time/Steal (s)	
0 GPU	2MBB	1831	115.18	0.063		N/A		26.57%
	3MBB	226	3.59	0.016	3351813	5.69	0.0000017	3.96%
1 GPU	2MBB	1554	101.81	0.065		N/A		32.02%
	3MBB	151	5.83	0.038	3112522	4.980	0.0000016	4.23%
2 GPUs	2MBB	1459	110.34	0.075		N/A		36.64%
	3MBB	166	2.54	0.015	3845257	6.15	0.0000016	4.75%
4 GPUs	2MBB	1090	96.13	0.088		N/A		40.72%
	3MBB	199	3.43	0.017	3955168	6.72	0.0000017	5.97%

Table 3.2: 2MBB: time taken for inter-node steals

that the 3MBB approach allows us to push the distributed system to its extremes by taking as much benefit as possible from different levels of parallelism of computing environments.

3.6 Conclusion

In this chapter, we have proposed and investigated dynamic load balancing schemes for parallelizing time-intensive B&B algorithms in node-heterogenous systems, where multiple CPUs and GPUs with possibly different properties are used. We designed an adaptive dynamic load balancing scheme for Multi-CPUs Multi-GPUs. Then we extend it to Multi-cores systems.

- **Multi-CPUs Multi-GPUs B&B (2MBB)**. This approach deals with two-level parallelism allowing for (i) distributed subtree exploration among PUs

and (ii) concurrent operations between every single GPU host and device. An adaptive scheme for sharing works based on PUs' computing power is also proposed. Through extensive experiments involving different PU configurations, this approach gives good performance at the scales up to 20GPUs and 128 CPUs. It's worth to highlight that to the best of our knowledge we firstly conduct experiments at the scale of 20 GPUs and 128 CPUs and the scalability of our approach is near optimal. However, the performance of this approach starts to drop significantly in larger scales compared with the linear speedup.

- **Multi-core Multi CPU Multi GPU B&B (3MBB)**. In this scenario where several GPUs are equipped with Multi-core CPUs, the hardware hierarchy is taken into account when balancing workload. This approach deals with two-level of hardware of compute nodes allowing for (i) intra-node steals performed by local threads of compute nodes to fully explore their locality (ii) inter-node steals performed by master threads of compute nodes through distributed memory systems. Furthermore, prioritizing intra-node steals over inter-node steals helps to speedup the computational portion while limits the communication over expensive links. Besides, a solution to split work pool into private and public portions for minimizing locking mechanism while performing intra-node steals is presented. Experimental result shows that the 3MBB improves up to 50% the results obtained using 2MBB.

We also showed that the B&B granularity can have a big impact on performance. Despite our approaches obtain near linear speedups at reasonable scales, there still might be room for improvements when considering fine-grained B&B instances and large scale heterogeneous systems. In particular, designing new distributed protocols in oder to fully take advantage of the power offered by the GPU is worth to be investigated in the future. Furthermore, the lessons learned from this study should help the design of new distributed and parallel B&B algorithms taking into account other types of compute devices such as the ones integrating many more compute cores with specific shared memory properties.

Besides, we confirmed that communication cost plays a crucial role in distributed computing environments while executing parallel B&B. In this chapter, we studied the impacts of intra-node and inter-node communication of Multi-core systems where PUs communicate to each other over different links with different cost. Let us notice that the differences in communication cost might be more complicated in other distributed platforms. For instance, geographically distributed systems or grid platforms have more complex heterogeneity in communication. In the next chapter, we will make a more in-depth study on the impact of link-heterogeneity on the performance of parallel B&B.

Chapter 4

Dynamic Load Balancing for Distributed *Link-Heterogeneous* Computing Environments

Contents

Main publication related to this chapter

T. Vu and B. Derbel. Link-Heterogeneous Work Stealing. *14th International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2014*, pages 354-363, Proceedings, IEEE, 2014.

4.1 Introduction

Heterogeneity in computing environments has recently appeared with grid computing [Dongarra 2006], volunteer and global computing [Korpela 2001, Fedak 2001] and more recently in cloud computing [Zhai 2011]. In contrast to dedicated supercomputers, such distributed environments are constituted of several components with variable degrees of heterogeneity. Node-heterogeneity with computing power, or different computing models was discussed in details in Chapter 3. Link-heterogeneity appears as different processing units (PUs) are connected by different communication networks [Kielmann 2002, Plaat 1999]. This chapter is devoted to the study of the impact of link-heterogeneity on dynamic load balancing algorithms for irregular tree search applications like B&B.

In parallel B&B algorithms, there exist many recent works [Mezmaz 2007a, Bendjoudi 2012a, Djamai 2013] solving challenging optimization instances on grid platforms. However, they did not focus on link-heterogeneity while trying to achieve high performance. In fact, the irregularities of B&B and the link-heterogeneity greatly complicate the challenges of achieving high performance. The irregularities continuously cause workload unbalanced situations which make dynamic load balancing algorithms function consecutively during the execution. The more unbalanced workload of a system is, the more load balancing operations are performed. However the cost of load balancing operations refers to how fast the workload is evenly distributed among PUs and it strongly depends on the underlying networks. If the networks are fast, the cost is cheaper since the workload is quickly distributed and unbalanced situations are solved rapidly. If the networks are slow, then the cost is more expensive because workload unbalance is not quickly fixed causing poor performance. Therefore, in link-heterogeneous systems where both fast and slow networks exist, handling the link-heterogeneity issues of the underlying networks is crucial to ensure the high performance of irregular applications like B&B. In fact, as the gap between computation and communication is substantial, it is likely that the parallel efficiency of the dynamic load balancing protocols drops significantly

Previous research studies have mostly considered the heterogeneity in communication speed by addressing the specific hierarchy proper to each computing environment such as multi-cpu multi-cluster platforms [Baldeschwieler 1996], geographically distributed multi-cluster multi-site grids [Van Nieuwpoort 2001, Van Nieuwpoort 2004, Janjic 2013], and others [Gast 2010, Pilla 2012, Acar 2000]. Despite that skillful design practices have been gained, previous studies targeted to specific computing contexts and ended up with a number of protocol variants which are essentially

platform-dependent. In particular, there is still little insights [Pilla 2012] into how network latency impacts fine-grain parallelism and how distributed communications have to be optimized to face the increasing complexity of heterogeneous networked resources in a portable and unified way. This makes it more complicated for programmers to deal with different link-heterogeneous of different distributed computational environments, which may result in ad-hoc implementations burdening the parallelization process and leading to non-efficient protocols. On the other hand, a knowledge about the computing platform could not be available at the time an application needs to be effectively deployed. For instance, clouds have the distinct characteristic of hiding the actual physical mapping of resources, and recent studies [Zhai 2011] showed that the interconnection latencies in modern virtualized environments pose the most severe obstacles when executing HPC workloads.

In this chapter, we will describe our generic approach to deal with dynamic load balancing for link-heterogeneous computing platforms. The proposed approach is generic in sense that it can be deployed in different distributed computing contexts exposing different properties in terms of communication latency. The remainder of the chapter is organized as the following. In Section 4.2, we present our classifications for current distributed geographical platforms. We then propose a simple model to abstract the complex of the link-heterogeneous platforms and review several state-of-the-art variants of work stealing to deal with the considered issues. In Section , we describe in detail our new proposed work stealing algorithm. In Section 4.4 we discuss the selected experimental method as well as all the link-heterogeneous platforms considered for our experiments. In Section 4.5 we present our experimental results and give insights into the dynamics of the proposed protocol. In Section 4.6, we finally conclude the chapter and raise some open research issues.

4.2 Work stealing in distributed link-heterogeneous computational environments

4.2.1 Distributed Link-heterogeneous Computational Environments

In link-heterogeneous computing environments, different network links offer different communication latencies which directly influence the performance of an application running on such platforms. Although the platforms are very complicated causing many obstacles for running HPC applications effectively, they are still being used to solve challenging applications requiring large computing resources of such platforms. Currently, there are many geographically distributed computing platforms exposing many link-heterogeneous settings, but they can be classified in two types according to their communication hierarchy:

- **Grid**. In the first classification, compute nodes are organized into groups according to the two-level communication hierarchy. In the first level, compute nodes of the same group communicate each other through link-homogeneous local area networks (LANs). In the second level, compute nodes of different

groups are interconnected by different link-heterogeneous wide area networks (WANs). For instance, Grid'5000 [Grid'5000] is an experimental testbed composed of a set of clusters distributed over 11 sites located in 11 different towns in France.

- **Peer-to-Peer**. In the second classification, the level of communicate hierarchy is more complicated. The compute nodes are interconnected by different WANs links with many levels of communication hierarchy. For instance, compute nodes are geographically distributed in different cities of the same country, different countries of the same continent or different continents.

Besides, different computing systems associated with different network configurations preserve different properties which makes it difficult to design a common model. Some works tried to propose a general model so that any network configuration can be instantiated from it. However modeling the link-heterogeneity of modern multi-computer distributed systems is a difficult task which is, *per se*, the subject of several dedicated research investigations, e.g. [Cappello 2005]. In our work, we also consider to build an abstract model allowing us to construct a generic algorithm which is independent of any particular distributed platforms. In line with previous studies [Beaumont 2010b], we focus on the case of distributed nodes having identical computing power and heterogeneous communication resources. The set of computing nodes V are fully connected and form a complete interconnection graph G, i.e., every node can communicate with any other node in the system by message passing. To model the interconnection heterogeneity between nodes, we endow the graph G with a function $\omega : V \times V \rightarrow \mathbb{R}$; which assigns for each pair of nodes i and j a real-valued weight $\omega_{i,j}$ informing about the cost experienced by node i when communicating with node j. The more the communication over edge (i, j) is costly, the higher is the weight $\omega_{i,j}$. Every node i is assumed to know solely the local weights $\omega_{i,j}$ connecting it with every other nodes $j \neq i$; thus hiding all the architectural characteristics the physical resources. In this study, we concentrate on link heterogeneity so that the function ω shall simply be viewed as a measure of nodes pairwise latency; that is, the network delay in a point-to-point message exchange.

Our model exposes only a flat view of the distributed environment. This is however sufficient to reason about link heterogeneity and to design an effective and efficient generic load-balancing protocol. Later in the chapter, different networked scenarios, ranging from purely virtual (peer-to-peer) environments to grid platforms, will be considered by simply instantiating the local weights $\omega_{i,j}$ at every distributed node in a consistent manner.

4.2.2 Work stealing in link-heterogeneous environments

In this section, we present some state-of-the-art work stealing algorithms dealing with link-heterogeneous distributed platforms. Since network links are no longer homogeneous, different links preserve different costs in terms of delay for stealing

messages while going through them. A bad network link offers a high cost and a long time for delivering stealing messages/responses. A good link offers a small cost and a short time in delay for sending the messages. Therefore, most of the proposed work stealing algorithms tried to minimize the cost of stealing messages by limiting the number of the messages sent through bad network links. In fact, a good work stealing algorithm for link-heterogeneous systems has to ensure a good workload balance by mostly profiting all the best network links. Let us remind that one of the most important questions in work stealing is *how to choose a victim?*. The random work stealing (RWS) presented previously in this thesis uses a uniform probability to choose a victim. It only considers all compute nodes in the same way, no matter where they are or how are the network links connecting them. It therefore can not be a good choice for link-heterogeneous computing platforms. In the following, we will briefly describe some existing algorithms leveraging RWS in this heterogeneous context.

4.2.2.1 Probabilistic Work Stealing (PWS)

This approach is an improvement of RWS while taking into account communication cost between compute nodes. The key idea of PWS is to pick up nearby compute nodes in priority. It suggests to use a measure to estimate the distance between computing nodes, and to modify the classical RWS algorithm in the following way: the probability to choose a target computer for steal attempts is not uniform anymore but instead proportional to the inverse of the distance between the thief and the target. With respect to our distributed model, this corresponds to every thief i choosing its victim with probability:

$$p_{i,j} = \frac{\frac{1}{\omega_{i,j}}}{\sum_{j \neq i} \frac{1}{\omega_{i,j}}} \tag{4.1}$$

The idea behind PWS is to privilege stealing over fast links without discarding the possibly slow links, in an attempt to reduce the average latency of steal requests. PWS has the nice property of being inherently local. This enables to capture the possibly different levels of hierarchy that might be implied by the computing architecture without loosing in generality nor in efficiency, as experimentally shown in [Quintin 2010] on a hierarchical system of 8 processors. Nevertheless, we found no experimental studies investigating the performance of PWS on large scale and more complex platforms. As systems' scale increases, it is likely that the gap between communication costs over different links increases substantially. This might has the effect of decreasing drastically the probability of stealing over slow links; thus eventually isolating some compute nodes and making work stuck at few regions without being able to flow fairly and quickly in the system.

4.2.2.2 Cluster-aware Hierarchical Stealing (CHS)

Basically, the idea of this algorithm is based on building a hierarchy of compute nodes. In this algorithm, all compute nodes are structured in a tree, and a thief

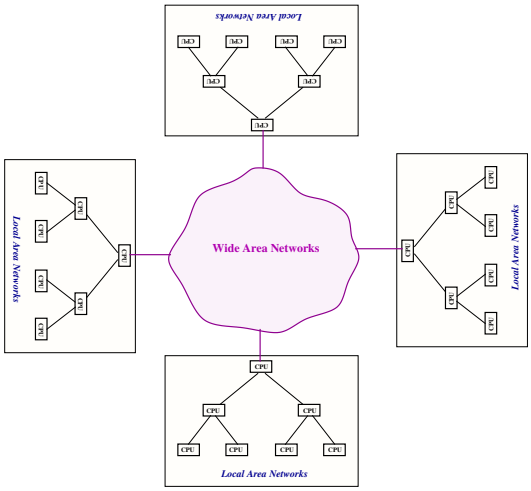

Figure 4.1: Tree Structure for Cluster-aware Hierarchical Stealing in Grid Platforms

looks for works among its children in the tree. When the whole subtree is idle, then the thief tries to steal work from its parent. This approach is mostly used in grid platforms, when several separate trees are built for several clusters and all the clusters are interconnected at their roots node through WANs.

This algorithm aims to minimize steal attempts through expensive WANs since the root node of a cluster only attempts to steal works from other clusters when it finds no work in its own cluster. The structure of the tree for grid platforms is sketched in Figure 4.1 and stealing mechanism along the tree is presented in Algorithm 8.

However, it is reported in [Van Nieuwpoort 2001] that this algorithm suffers some shortcomings. The whole cluster is stalled while the root of the corresponding tree tries to steal works via WANs. Therefore, this approach is not suitable for high irregular applications like B&B.

4.2.2.3 (Adaptive) Cluster-aware Random Stealing

The Adaptive Cluster-aware Random Stealing protocol is an improvement of the so-called Cluster-aware Random Stealing (CRS) protocol described originally in [Van Nieuwpoort 2001]. As indicated by their names, both protocols, were designed specifically for grid platforms with two-level hierarchical communication. The only assumption made in order to work properly is that each node is explicitly given a full knowledge about locations of other nodes in the system. In other words, a node has two sets of neighbors, one set of local neighbors containing all nodes locating in the same cluster, one set of remote neighbors comprising all nodes locating in other clusters.

CRS [Van Nieuwpoort 2001]. This algorithm extends RWS by allowing each node to steal works from both local neighbors and remote neighbors. In general, an idle node performs two steals, one to local neighbors and one to remote neighbors.

Algorithm 8: Cluster-aware Hierarchical Stealing in Grid Platforms

```
1  flag_wide_area_steal_request ⟵ false ;
2  while termination do
3      job ⟵ check for work from local pool ;
4      if job ≠ ∅ then
5          process job ;
6      else
7          C ← set of my children nodes in the tree;
8          if all nodes of C already sent steal requests to me then
9              if I am the root then
10                 if flag_wide_area_steal_request= false then
11                     flag_wide_area_syn_steal_request ⟵ true ;
12                     R ← set of root nodes of other clusters;
13                     r ← a random node of R;
14                     work ← Send a steal request to r;
15             else
16                 p ← parent node;
17                 work ← Send a steal request to p;
18         else
19             c ← a random node of C;
20             work ← Send a steal request to c;
21         if work ≠ ∅ then
22             Unpack work and push into the local pool;
23             if work is stolen from another root then
24                 flag_wide_area_steal_request ⟵ false ;
```

The two steals are sent asynchronously hence the idle node does not stay blocked waiting for a response. If it receives a reject message from either local neighbors or remote neighbors and it is still idle then it will continue to steal from them. Whenever a node is idle, it first attempts to steal from the local neighbors through LANs, and *in parallel* it sets a flag and further sends an additional steal requests through WANs to another node of its remote neighbors. The flag is reset whenever it receives a response (reject or works) from remote neighbors. As long as the flag is set, idle nodes only steal works from its local neighbors. This approach is detailed in Algorithm 9.

The extensive study conducted in [Van Nieuwpoort 2001] shows that CRS provides very good performances; however, it still suffers from the following limitations:

- Knowledge of grid platforms should be made available to all available compute nodes.

Algorithm 9: CLUSTER-AWARE RANDOM STEALING

```
Data: L_{\i} = {1, 2, ··· , n} \ {i}: local neighbors' identifiers;
      R = {n + 1, n + 2, ··· , n + m}: remote neighbors' identifiers;
1  flag_asyn_steal_request ⟵ false ;
2  flag_syn_steal_request ⟵ false ;
3  while termination do
4      job ⟵ check for work from local pool ;
5      if job ≠ ∅ then
6          process job ;
7      else
8          if flag_syn_steal_request=false then
9              flag_syn_steal_request ⟵ true ;
10             u ← a random node of L_{\i};
11             Send a steal request to u;
12         if flag_asyn_steal_request=false then
13             flag_asyn_steal_request ⟵ true ;
14             r ← a random node of R ;
15             Send a steal request to r;
16         HANDLE_STEALS() ;
```

Procedure HandleSteals

```
1  if a reply msg from node ℓ is pending then
2      work ⟵ check for work by processing msg;
3      if work ≠ ∅ then
4          Unpack work and push into the local pool;
5      if ℓ is a local neighbor then
6          flag_syn_steal_request ⟵ false ;
7      else
8          flag_asyn_steal_request ⟵ false ;
```

- The protocol is designed for grid systems with two-level communication hierarchy and it raises open questions about how to adapt it to other platforms with more complex communication hierarchy.
- When both steals are successful, tasks might be unnecessary transferred from one cluster to another which causes ping pong effects of moving tasks back and forward between clusters.
- This approach is only suitable for link-homogeneous WANs as stealing requests through WANs are *randomly* performed. In fact it brings the same limitation as in the RWS.

ACRS [Van Nieuwpoort 2004]. This algorithm extends on the CRS by adapting the non-uniform probability given by Equation 4.1 when choosing remote neighbors: the probability of choosing a remote neighbor is inversely proportional to the communication cost of the connected WANs between them. Generally speaking, ACRS uses RWS when stealing works from local neighbors and PWS when stealing works from remote neighbors. Despite ACRS improves CRS, it still does not address the three previously mentioned limitations.

4.3 Link-heterogeneous work stealing

Though there are several differences in principles and designs, all the above protocols have a common aspect. They all attempt to increase the locality while balancing workload in link-heterogeneous systems by preferring to steal works from *close neighbors* through good links rather than *remote neighbors* through bad links. This is the key ingredient to counteract the link-heterogeneous issues. In PWS, the *remote neighbors* are set with relative small probability to be chosen resulting in only few steal messages spend longer time to across bad links. Similarly, in hierarchical work stealing, the *remote neighbors* are only chosen when no work is found locally inside a cluster and the root node is in charge of stealing works from *remote neighbors*. In contrast, the CRS and ACRS try to hide the communication cost when stealing works from remote neighbors by stealing works *asynchronously* both from *local neighbors* and *remote neighbors* at once. In fact, all the protocols bring many advantages but also suffer from several disadvantages as presented in the above sections. All of these aspects inspired us to design a protocol inheriting all the advantages and overcoming the disadvantages.

Assuming that a distributed link-heterogeneous platform is viewed as a complete weighted graph G, we can make the two following observations. The first one is that the more we encourage nodes to communicate over fast links according the constructed steal probabilities over G, the more efficient woks are offloaded. The second observation is that the stealing preference to fast links might not be efficient enough if most of the workload stays on regions connected by slow links. These observations pose a dilemma which is difficult to face because of the unpredictable nature of workload.

For a node to distributively take the good decision at runtime, we propose the generic distributed algorithm called Link-Heterogeneous Work Stealing (LWS) depicted in Algorithm 10. Our approach is able to effectively detect *local neighbors* and *remote neighbors* at run time which makes it less dependent on distributed computing platforms. The key ingredients of LWS are summarized as the following:

- Identify a set of preferred victims by synchronous steals. The synchronous steals are performed according to a modified PWS such that the stealing probability is calculated based on both physical distance and logical distance among compute nodes. The physical distance represents how much time it takes for them to communicate each other physically. The logical distance represents

how data is close between thieves and peered victims.

- Every node splits its neighbors into local neighbors and remote neighbors using a standard classification algorithm, namely, the K-means algorithm [Dunham 2002].
- Asynchronous steals are only enabled when necessary in order to avoid the ping-pong effects.

In the following, we give a detailed description of our LWS algorithm for load balancing irregular workload in link-heterogeneous environments. We recall that the high level code of LWS is depicted in Algorithm 10.

Synchronous Steal. When becoming idle, a node first starts sending *synchronous* steals. A thief sends synchronous steals probabilistically as in PWS; but using a modified probability function to select victims (lines 18 to 22):

$$p_{i,j} = \frac{s_{i,j} \cdot r_{i,j}}{\sum_{\ell \neq i} s_{i,\ell} \cdot r_{i,\ell}} \tag{4.2}$$

where:

$$s_{i,j} = \frac{\frac{1}{\omega_{i,j}}}{\sum_{j \neq i} \frac{1}{\omega_{i,j}}} \qquad r_{i,j} = \frac{c_{i,j}}{\sum_{\ell \neq i} c_{i,\ell}}$$

In fact, every node i stores the number of synchronous work requests $c_{i,j}$ that have been issued towards node j (lines 3 and 21). The probability for node i to choose victim j is inversely proportional to the communication latency $\omega_{i,j}$ and proportional to the local counter $c_{i,j}$. This probability is denoted by variable $p_{i,j}$ (line 13) which is computed as a multiplicative aggregation of the two probability functions $r_{i,j}$ (line 12) and $s_{i,j}$. Clearly, this strategy accentuates the locality between a thief i and its *previous* victims, and aims at isolating few very preferred victims from where it is likely to be very fast to check for work.

Victim Partition. The victims which are likely to be connected with slow links, are not completely discarded. They are in fact requested for work *asynchronously* as in ACRS to avoid loosing time waiting for steal replies. A thief needs to separate between preferred synchronous victims and asynchronous targets using a partitioning procedure (line 10). Procedure PARTITION_VICTIMS is actually a simple implementation of the well known k-means algorithm, e.g., [Dunham 2002]. For $k = 2$, the 2-means algorithm is a heuristic to cluster a set of objects in two groups; such that, variability within the same group is minimized and variability between the groups are maximized. In the design of LWS, victims' counters are used to define variability. The group of victims having the lowest counters values is identified as

Algorithm 10: Link-Heterogenous Work-Stealing (LWS): distributed high level pseudo-code for every node $i \in V$.

Data: $V_{\setminus i} = \{1, 2, \cdots, n\} \setminus \{i\}$: neighbors' identifiers;
$\Omega_i = \{\omega_{i,j}, \forall j \in V_{\setminus i}\}$: latencies between i and j;
$C_i = \{c_{i,j}, \forall j \in V_{\setminus i}\}$: counter values of j stored at i ;
T: a parameter ;

1 flag_asyn_steal_request $\longleftarrow$ $false$;
2 flag_syn_steal_request $\longleftarrow$ $false$;
3 $\forall j \in V_{\setminus i}$, $c_{i,j} \longleftarrow 1$; $X_i \longleftarrow 0$; $Y_i \longleftarrow 0$;
4 **while** *termination* **do**
5 job $\longleftarrow$ check for work from local pool ;
6 **if** *job* $\neq \varnothing$ **then**
7 process job ;
8 **else**
9 **if** $\sum_{j \in V_{\setminus i}} c_{i,j}$ % $T = 0$ **then**
 `// Update asyn. victims`
10 $V_{\text{asyn}} \longleftarrow$ **VICTIM_PARTITION()** ;
 `// Update selection probabilities`
11 $\forall j \in V_{\text{asyn}}$, $q_{i,j} \longleftarrow \frac{1/\omega_{i,j}}{\sum_{\ell \in V_{\text{asyn}}} 1/\omega_{i,\ell}}$;
12 $\forall j \in V_{\setminus i}$, $r_{i,j} \longleftarrow \frac{c_{i,j}}{\sum_{\ell \in V_{\setminus i}} c_{i,\ell}}$;
13 $\forall j \in V_{\setminus i}$, $p_{i,j} \longleftarrow \frac{s_{i,j} \cdot r_j}{\sum_{\ell \in V_{\setminus i}} s_{i,\ell} \cdot r_\ell}$;
 `// Syn. and Asyn. Steal attempt(s)`
14 **if** *flag_asyn_steal_request*=$false$ && $Y \geq X$ **then**
15 $s \leftarrow$ a node in V_{asyn} selected with prob. $q_{i,s}$;
16 **Send** an asynchronous work request to s ;
17 flag_asyn_steal_request $\longleftarrow$ $true$;
18 **if** *flag_syn_steal_request* = $false$ **then**
19 $k \leftarrow$ a node in $V_{\setminus i}$ selected with prob. $p_{i,k}$;
20 **Send** a synchronous work request to k ;
21 $c_{i,k} \leftarrow c_{i,k} + 1$;
22 flag_syn_steal_request $\longleftarrow$ $true$;
23 HANDLE_TIMER() ;

the set V_{asyn} from which asynchronous steals should be performed.

Asynchronous Steal. The asynchronous steal mechanism used in LWS (lines 14

Procedure VictimParition

Result: The set of asynchronous target victims.

```
// 2-means based on steal counters
```
$j_1 \longleftarrow$ a node from $V_{\setminus i}$ uniformly at random ;
$j_2 \longleftarrow$ a node from $V_{\setminus i} \setminus \{j_1\}$ uniformly at random ;
$t \longleftarrow 0$; $m_1^t \longleftarrow c_{i,j_1}$; $m_2^t \longleftarrow c_{i,j_2}$;
repeat
  $t \longleftarrow t+1$;

  `// Assignment step`
  $V_1 \longleftarrow \left\{j : \left|c_{i,j} - m_1^{t-1}\right| \leq \left|c_{i,j} - m_2^{t-1}\right|\right\}$;
  $V_2 \longleftarrow V_{\setminus i} \setminus V_1$;

  `// Means Update`
  $m_1^t \longleftarrow \frac{1}{n-1} \sum_{j \in V_1} c_{i,j}$;
  $m_2^t \longleftarrow \frac{1}{n-1} \sum_{j \in V_2} c_{i,j}$;
until $m_1^t = m_1^{t-1}$ *&&* $m_2^t = m_2^{t-1}$;
if $m_1^t < m_2^t$ **then return** V_1; **else return** V_2;

Procedure HandleTimer

Result: Handles work replies and updates 'timers' for asynchronous steals.

if *a reply* msg *from node* ℓ *is pending* **then**
  work $\longleftarrow$ check for work by processing msg;
  if *work* $\neq \varnothing$ **then**
    Unpack work and push into the local pool;
  if $\ell = k$ **then**
    `// Reply w.r.t the synch. steal`
    if *work* $\neq \varnothing$ **then**
      $Y_i \longleftarrow 0$;
      $X_i \longleftarrow X_i + \omega_{i,k}$;
    else
      $Y_i \longleftarrow Y_i + \omega_{i,k}$;
      $X_i \longleftarrow X_i/2$;
    flag_syn_steal_request $\longleftarrow false$;
  else
    `// Reply w.r.t the asynch. steal`
    flag_asyn_steal_request $\longleftarrow false$;

to 22) differs from ACRS in the following aspect. As implemented by the second condition of the if instruction line 14, an asynchronous work steal is *not* systematically

issued as soon as a node becomes idle. In order to avoid an unnecessary work transfer, a thief i first makes a number of synchronous steal attempts towards its preferred victims. The starting signal before issuing an asynchronous steal is handled in procedure HANDLE_TIMER through control variables X_i and Y_i. These variables play the role of *adaptive* timers. The idea is to distributively *detect* the availability of work among nearby processors by self-adjusting a time window over which work, if any, is expected to flow synchronously. Inspired by the additive-increase/multiplicative-decrease feedback approach for congestion avoidance [Chiu 1989], if a synchronous steal made by thief i to preferred victim k is successful then the waiting window X_i is increased proportional to network latency, that is by $\omega_{i,k}$. Otherwise, X_i is decreased by half and the 'elapsed time' Y_i is increased by $\omega_{i,k}$. The formulas are summarized in Equation 4.3 and Equation 4.4. Only after Y exceeds X that an asynchronous steal is sent to a victim s selected from V_{asyn} with probability q_s (line 15), inversely proportional to the communication costs.

$$X_i(t+1) = \begin{cases} X_i(t) + \omega_{i,k} & \text{if synchronous steal to } j \text{ is successful} \\ X_i(t) \cdot \dfrac{1}{2} & \text{if synchronous steal to } j \text{ is NOT successful} \end{cases} \tag{4.3}$$

$$Y_i(t+1) = \begin{cases} 0 & \text{if synchronous steal to } j \text{ is successful} \\ Y_i(t) + \omega_{i,k} & \text{if synchronous steal to } j \text{ is NOT successful} \end{cases} \tag{4.4}$$

In Algorithm 10, we remark that in order to avoid introducing computing overheads (i.e., victim partitioning procedure), some control variables are only updated periodically. This is ruled by parameter T defining the number of synchronous steal attempts that have to be performed before control variables are updated (line 9).

Termination and Knowledge sharing. These are two important issues of LWS which have not been discussed yet. As similar to other work stealing approaches presented in the previous chapters, we still use a binary tree to distributively handle the two issues. The termination is detected in the 'Up-Down' distributed manners and it takes at most $log_2 N$ hops between the leaves and the root. Similarly, the knowledge sharing takes $2 * log_2 N$ hops to update a new found better solution so far to all available compute nodes in the systems allowing an effective exploration of the parallel B&B.

To conclude, let us emphasis that load balancing in LWS is achieved distributively by every node in a purely local manner without needing any global information about the system hierarchy or network construct: Only the local communication costs $\omega_{i,j}$ are exploited locally by every thief i. Although LWS is generic, it introduces advanced control operations which are expected to provide better performances. The rest of the chapter is devoted to the experimental evaluation of LWS.

4.4 Experimental Design and Methodology

Our experiments are designed not only to assess the performance of LWS but also to show the impact of link heterogeneity and application granularity. Therefore, we analyze all competing protocols for two different applications and study different network scenarios in which the communication hierarchy and the range of latencies varies from two-level to a more complex distributed setting. We start describing the methodology we adopt in our experimental design.

4.4.1 Methodology

Currently there are three common categories for experimental evaluation: real experiments, simulation, and emulation. Real experiments involve running a real-application on a real experimental-platform, which is generally believed to provide high realism. However, experimenters face many difficulties to validate their findings, e.g., platform dependency, result reproducibility, etc. In contrast, simulation facilitates complex configurations at the prices of a relative loss in realism; also, the applicability of real applications could be questionable. In this thesis, we want to consider different complex network configurations and to have high confidence in collected results. By combining the advantages of the two first approaches, emulation appears to be an appropriate solution for experimenting complex distributed configurations with real applications. In this chapter, we decided to use Distem [dis , Sarzyniec 2013] to emulate a broad range of link-heterogeneous network configurations. Distem is a distributed systems emulator for realistic environments appearing in cloud, peer-to-peer, high performance computing or grid systems. It uses virtualization to transform a homogeneous real-cluster into an experimental platform where nodes have different power and/or are linked together through a complex network topology; thus making it an ideal tool for our study. We would like to emphasis that real-CPU compute nodes of a real distributed test-bed are used in our experiments; only network link latencies are artificially configured through Distem.

4.4.2 Network instances

As discussed in Subsection 4.2.1, there exists many possible levels in communication hierarchy of geographically distributed computing platforms. In our classification, we classify them into two types such as Grid with two-level communication hierarchy and P2P with a complex of many-levels communication hierarchy. In order to get more insights into the impact of link-heterogeneity on performance, we consider Grid as well as the P2P compute settings with different network heterogeneity.

4.4.2.1 Grid or Clustered environments (C_Env)

We consider a grid computing context, where a number of homogenous clusters are available in different geographically distributed sites connected with different speeds.

More precisely, a two-level communication hierarchy is considered. In the first level, the latency between nodes from the same cluster is set to a fixed value of 0.2 ms. In the second level, clusters are fully connected with WAN links which are split into two sub-groups, fast WAN links with latency in the range $R^{\text{grid}}_{\text{fast}} = \{30, 40, 50\}$ (ms); and slow WAN links with latency in the range $R^{\text{grid}}_{\text{slow}} = \{100, 150, 200\}$ (ms). To keep the experiments manageable, the number of computing nodes is fixed to 128 and the following variations are considered when setting this type of distributed environment:

- The number of clusters c is fixed to 8 with the same number of nodes in each cluster. The latency between pairwise clusters are randomly drawn from one of the two ranges $R^{\text{grid}}_{\text{fast}}$ and $R^{\text{grid}}_{\text{slow}}$ following a bernoulli-distribution of parameter $p \in \{0, 0.25, 0.5, 0.75, 1\}$, i.e., latency is picked uniformly at random from $R^{\text{grid}}_{\text{fast}}$ (resp. $R^{\text{grid}}_{\text{slow}}$) with probability p (resp. $1 - p$). We shall use notation C_ENV$(c = 8, p)$ when referring to a randomly generated network instance in this setting.

- The number of clusters c varies in the range $\{1, 2, 4, 8, 16\}$ with equal number of nodes in each cluster. Network latency between clusters is uniformly distributed in the range $R^{\text{grid}}_{\text{fast}} \cup R^{\text{grid}}_{\text{slow}}$ (which corresponds to $p = 0.5$ in the previous scenario). We shall use notation C_ENV$(c, p = 0.5)$ when referring to a randomly generated network instance in this setting.

4.4.2.2 P2P or 'Virtualized' flat environments (VF_ENV)

Motivated by cloud, Internet, and peer-to-peer computing systems, we extend on the previous network setting in order to take into account more complex communication patterns. No explicit fixed hierarchy is set; Instead, compute nodes are fully connected and the communication latency between each pair of nodes are randomly drawn from one of the two ranges $R^{\text{flat}}_{\text{fast}} = \{1, 3, 5, 10\}$ (ms) and $R^{\text{flat}}_{\text{slow}} = \{50, 100, 150, 200\}$ (ms) following a bernoulli-distribution of parameter $p \in \{0, 0.25, 0.5, 0.75, 1\}$, i.e., latency is picked uniformly at random from $R^{\text{flat}}_{\text{fast}}$ (resp. $R^{\text{flat}}_{\text{slow}}$) with probability p (resp. $1 - p$). Generally speaking, the group of fast links is for example relative to compute nodes (or peers) in different clusters of different cities of same country, or different workstations in distant countries of the same continent. The group of slow links represents typically inter-continent communication links. We shall use notation VF_ENV(p) when referring to a randomly generated network instance in this setting.

4.4.3 Protocol deployment

The experimented protocols are: LWS, RWS, PWS, CRS and ACRS. Although RWS is known to perform worst than the other protocols, we still include it in our experiments in order to give a more comprehensive picture on relative performance. In fact, the performance of RWS is the worst case showing an upper bound for the

performance of our experiments. Notice also that ACRS degenerates to PWS and CRS degenerates to RWS for VF_Env.

Our experiments are performed using Distem as a network emulator running on one cluster of the Grid'5000 experimental platform [Grid'5000]. The cluster in use has 92 nodes, each one equipped with 2 CPU of 2.5 Ghz Intel Xeon processor with 4 cores per cpu and a network card infiniband-20G. Every compute node in our experiments is deployed with Distem on a dedicated physical compute core initialized and configured with the corresponding communication latencies. The latencies are managed internally by Distem without any additional operations at the application level. We also remark that LWS is deployed with parameter T, controlling the frequency of victim partitioning and steal probability update, empirically set to 100.

Like in the previous chapter of this thesis, we use the Flowshop instances as target problems in our experiments, as well as the UTS benchmark in order to gain in genericity.

4.5 Experimental Results and Analysis

We start by analyzing the differences in execution time between competing algorithms. Then-after, we give a more focused analysis of protocols' behavior.

4.5.1 Overall performances

In Fig. 4.2, we summarize the average execution time obtained for the different network settings and applications.

4.5.1.1 Protocols pairwise comparison

The basic RWS is the worst performing protocol, and the other competing algorithms are able to improve performance substantially. Zooming in the performance of LWS, ACRS and PWS, we found that LWS performs better with a variable gap depending on the type of faced heterogeneity. In average over all the runs made in C_Env, LWS saves 38% and 10% execution time (resp. 16% and 26%) when compared to ACRS and PWS for the UTS benchmark (resp. B&B). The gaps are even larger in VF_Env where LWS saves 15% and 43% execution time (resp. 62% and 65%) in average when compared to PWS and RWS for UTS (resp. B&B). We further performed several statistical tests to assess the superiority of LWS. By running Friedman tests, the null hypothesis that protocols' average execution times are same cannot be accepted in any setting with a high confidence level. By running pairwise post-hoc Mann-Whitney tests to evaluate the differences in execution times, we were able to report the following observations (where 'significance' is with respect to a p-value of at least 0.05):

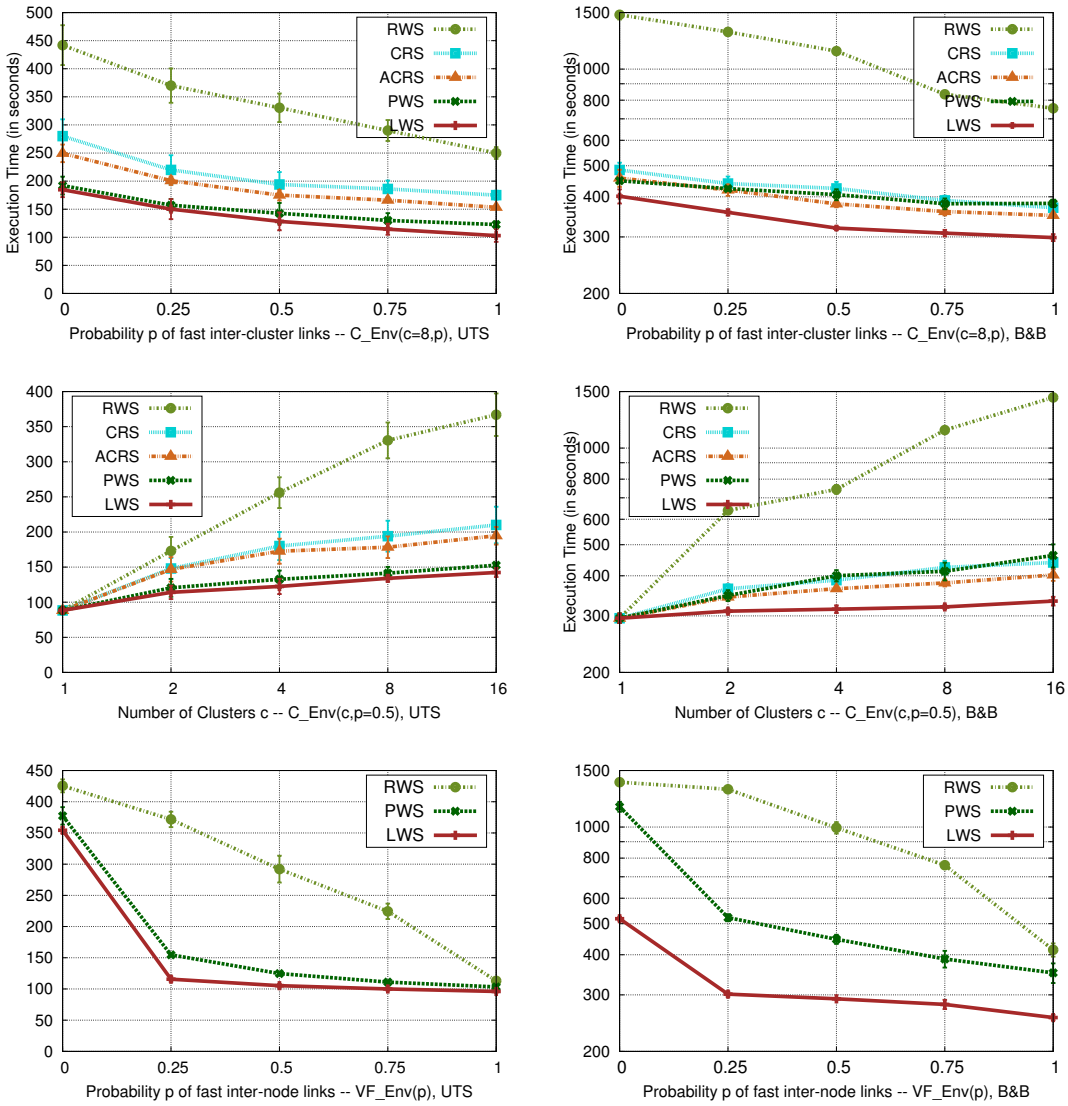

Figure 4.2: The obtained execution time of competing algorithm in different scenarios. **First column**: UTS. **Second column**: B&B. **From Top to Bottom**: C_ENV$(8, p)$, C_ENV$(c, 0.5)$, and VF_ENV(p). Error bars shows 95% confidence interval. Notice that because the gap between RWS and other protocols is relatively huge for B&B, we put the y-axis in the log scale for more readability.

In the C_ENV settings we found a significant difference between LWS and ACRS for both UTS and B&B. In fact, LWS is able to provide up to 58% gain in execution time. Comparing LWS to PWS, we are able to report significant superiority of LWS only for B&B where the gap in execution time is up to 26%. For UTS, while LWS performs slightly better than PWS in average, the difference was not statistically significant. This can be explained by the fact that work units in UTS take

much less time to be processed compared to the realistic B&B computations; thus making the impact of link-latency on the overall processing time more pronounced for B&B than for UTS. One can also notice the very good performances obtained with PWS compared to ACRS; the former being able to provide significantly better execution times for the UTS benchmark, but not for B&B.

In the VF_Env settings LWS performs significantly better than PWS and RWS; achieving up to an acceleration factor of two acceleration. The larger gaps in execution time are obtained with B&B rather than with UTS, which we attribute to the work granularity of B&B. Notice also the expected poor performance obtained with RWS which indicates that link-latency can have a deep impact on standard work-stealing.

4.5.1.2 Impact of heterogeneity level

Fig. 4.2 allows us to extract interesting observations on the impact of heterogeneity. In the C_Env$(8, p)$ settings, we see a clear tendency of the execution time of all protocols to increase when more slow inter-cluster links are included, i.e., for small values of probability parameter p. The same effect can be observed for C_Env(c,0.5) when the number of clusters increases, which is clearly because the average latency between compute nodes also increases. Notice however, the relative robustness of LWS for B&B with varying number of clusters. In the VF_Env(p) settings, we see a significant impact of link latency when parameter p is in the range $\{0, 0.25\}$. This corresponds to roughly less than 25% of links being fast. Below that percentage, LWS suffers a deterioration in execution time for UTS while being relatively robust for B&B. Above that percentage, both LWS and PWS stabilize quickly with LWS being better. This indicates that LWS is able to schedule work steals efficiently by exploiting maximally the few fast links available in the network.

4.5.2 Analysis of Protocol Dynamics

To better understand why LWS is performing better, we extracted several measures rendering its dynamics compared to PWS and ACRS; and providing new insights into how to deal with network heterogeneity.

4.5.2.1 LWS *vs.* PWS in VF_Env

One key difference between LWS and PWS is the probability of synchronous steals. In Fig. 4.3, we illustrate the values taken by the probability that node i selects a victim j in a synchronous steal attempt for all i and j. For clarity, we plot these values in the form of heat-maps; and we also show their empirical distribution in the Fig. 4.4. Since probabilities in LWS were observed to converge quickly to a stationary regime where they change only very marginally, data is extracted from the latest stages of protocol execution.

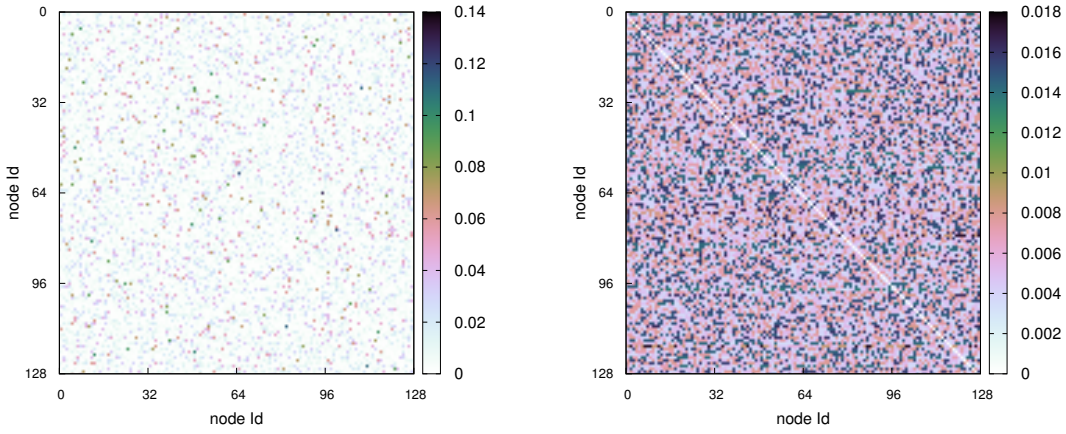

Figure 4.3: **Left** (resp. **Right**) Heat-map matrix of synchronous-steal probability of LWS (resp. PWS) in VF_ENV($p = 0.5$). The darker a point is, the higher is the probability. Results are for one typical run of B&B.

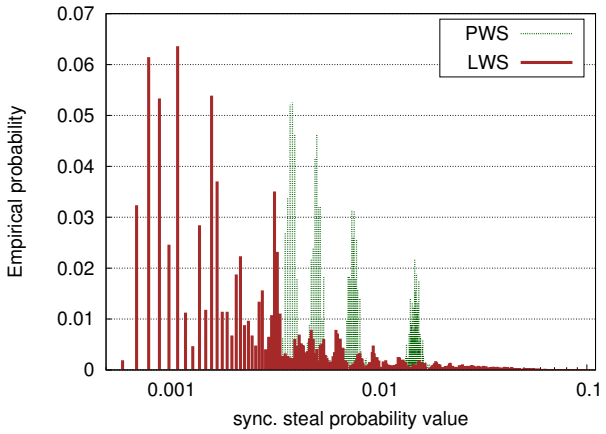

Figure 4.4: Empirical cumulative distribution of probability's values. Results are for one typical run of B&B. Results are for one typical run of B&B.

A good way to understand the implications of Fig. 4.3 is to remember that if RWS was considered, then steal probabilities would be a constant equals to $1/128 \simeq 0.0078$. We observe that a thief executing LWS has very few nodes where the probability of being selected for a synchronous steal is very high. In contrast, although the probabilities in PWS are not 'uniform', the gap between them is less pronounced than LWS. This means that there are a large number of victims in PWS that have roughly equal chances of being selected. By viewing victim selection probabilities as defining a random weighted graph [Janson 1998, Garlaschelli 2009], LWS is distributively electing very few preferred edges connecting nodes with a high prob-

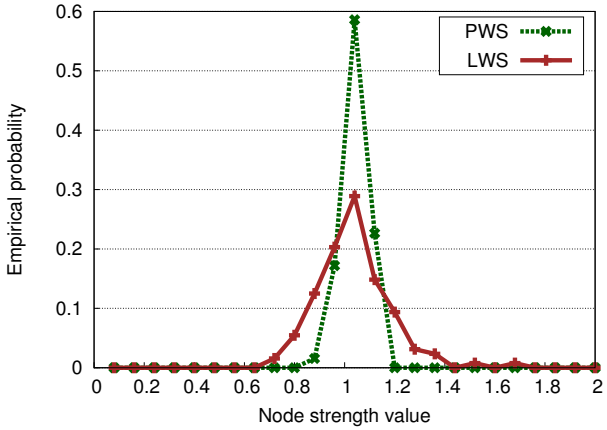

Figure 4.5: Strength empirical cumulative distribution of a typical run of LWS and PWS in VF_ENV($p = 0.5$) and UTS.

ability. These edges are inducing a probabilistic overlay structure allowing work to flow distributively. The so-constructed overlay is very sparse in the case of LWS while it is not in the case of PWS; and we suspect it to implicitly induce very short paths between nodes in terms of network latency. In Fig 4.5, we also show the empirical cumulative distribution of the so-called *Strength* 'feature' used in random graph theory [Garlaschelli 2009]. The strength of any node i is computed as the sum of the probabilities that other nodes j select i in a synchronous steal. The strength of node i informs about the number of work steals that i should expect if all other nodes were idle. Remark that the strength of any node would be exactly 1 in the case of RWS. From Fig 4.5, we can see that the strength distributions induced by LWS and PWS look both like a gaussian of mean 1; but with a higher variance for LWS. This indicates that there are some nodes in LWS that are acting like 'Hubs' from where work is likely to transit and to flow out more frequently.

To summarize these observations, we can say that LWS is distributively constructing a kind of probabilistic network spanner [Peleg 1989] connecting nodes. This structure has the very specific properties of being sparse and containing very few nodes with high in-degree; thus improving work locality and optimizing the global cost of synchronous work transfers.

4.5.2.2 LWS *vs.* ACRS in C_ENV

One key design choice made in LWS compared to ACRS is the time that an asynchronous steal attempt is performed. In Fig. 4.6, we compare the adaptive additive-increase/multiplicative-decrease strategy of LWS with the situation where an idle node would just wait for a *prefixed* period. The results depicted in Fig. 4.6 are obtained by running this new protocol variant with a range of waiting periods (ACRS corresponds to a waiting time being set to 0). Clearly, waiting time has an impact on performances: a node should neither send an asynchronous work request imme-

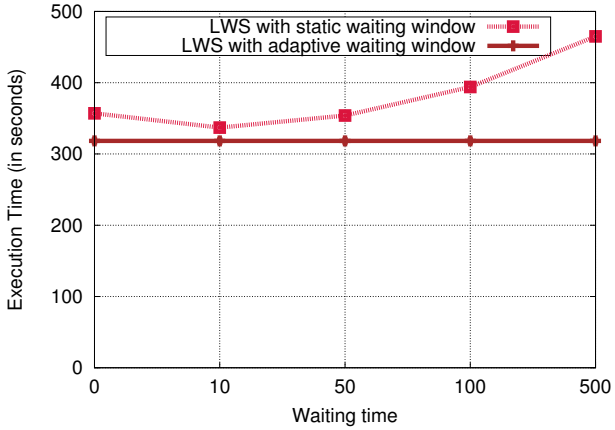

Figure 4.6: Comparison between adaptive LWS, and LWS with a fixed waiting period. Results are for B&B in C_ENV$(8, 0.5)$.

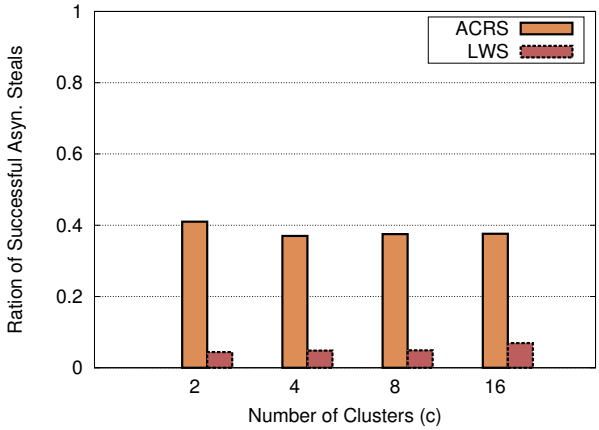

Figure 4.7: Ratio of asynchronous work transfers for C_ENV$(c, p = 0.5)$ and B&B.

diately, nor it should wait for a long period. Nevertheless, no pre-fixed value is able to outperform the adaptive strategy of LWS. We are then able to confirm that the dilemma of waiting for work to flow with synchronous steals or attempting to steal asynchronously is well solved by LWS. This property is obtained although every distributed node has a very flat view of the system.

In Fig. 4.7, we further show the ratio of asynchronous work transfers with respect to all steal attempts that a node performed. This ratio appears to be much smaller for LWS independently of the number of clusters available. This indicates that the adaptive mechanism of LWS tends to minimize unnecessary work transfers; whereas in ACRS work is likely to travel back-and-forth between clusters.

To conclude our analysis, the lifestory of a typical run of LWS is presented in Fig. 4.8 informing about the ratio of idle and working nodes. We found that work is evenly distributed among clusters. The ratio of idle nodes stays relatively low in this harsh network scenario (20% in average); only when approaching termination that

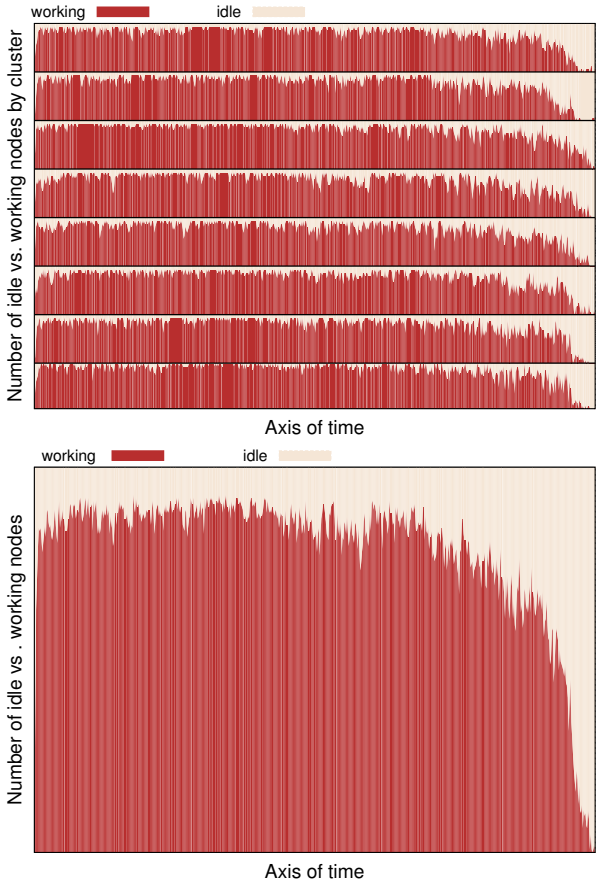

Figure 4.8: Lifestory of nodes in LWS. The x-axis refers to elapsed time until application termination. **Top** (resp. **Bottom**) figure summarizes the status of nodes in every cluster (resp. overall). Results are for one run of B&B in C_Env($c = 8, p = 0.5$); data was collected by normalizing and discretizing timestamps at every node.

parallel efficiency starts to drop which is inherent to the nature of the unbalanced tree search applications we are studying.

4.6 Conclusion

In this chapter, we addressed the question *how to choose a victim* at runtime in work stealing protocols while solving irregular applications like B&B on link-heterogeneous computing platforms. We described our solution, called Link-Heterogeneous Work Stealing (LWS), to address the heterogeneity of link latency in networked computing systems. LWS is generic, local, and self-adaptive. Although LWS is not designed to fit a specific network context, our experiments show that LWS is able to outperform well-established load balancing protocols. This is due to the accuracy of LWS in self-adjusting synchronous and asynchronous steals attempts, as well as the

advanced and self-optimized communication patterns induced among nodes. The lessons learnt from our analysis support that scheduling highly irregular applications in large scale networked platforms can be attained with generic unified mechanisms without neither complicating the underlying protocols nor loosing in efficiency.

In future work, we intend to study the behavior of LWS when network links experience speed variance during execution time. Since all LWS components are fully distributed, we expect LWS to react efficiently without further design efforts. Moreover, we only focused on heterogeneity in link latencies and relaxed the other sources of heterogeneity, e.g., compute power, throughput, etc. Combining LWS and the approach presented in Chapter 3 to take into account such complex settings which are likely to appear in real-world networked environments is also one of our future goals. Another difficult research direction would be to integrate LWS in existing cloud computing environments and to evaluate the obtained performances with respect to classical Grid and HPC facilities, in an attempt to push forward the landscape of possible parallel applications in virtualized platforms.

Conclusions and Perspectives

In this thesis, we presented our contributions in improving the performance of parallel Branch-and-Bound algorithms for solving Combinatorial Optimization Problems. In particular, the different approaches presented in this thesis address the three following challenging issues:

- The irregularity of parallel B&B.
- The heterogeneity in computation of computing resources.
- The heterogeneity in communication of communication resources.

In the first contribution, to solve the irregularity of Parallel B&B, we proposed a tree-based dynamic load balancing algorithm aiming at enhancing cooperation between processing when searching for work. The extensive experimental study showed that this approach is able to produce a good performance in large scales up to 1200 processing units while improving over the previous Master-Worker [Mezmaz 2007a] or Hierarchical Master-Worker paradigm [Bendjoudi 2012a]. The achieved speedup is obtained due to a comprehensive design of our approach. Firstly, processing units can communicate directly with their neighbors (e.g the children and parent) in the tree when stealing works. Secondly, direct communication of processing units belonging to different subtrees can be established through bridge edges in order to speedup their communication along the tree. Thirdly, a cooperative tree-dependent work sharing policy is proposed in order to reduce the communication time for stealing requests. This policy in fact allows to minimize the number of stealing requests as processing units not only fetch works for themselves but also for serving their neighbors of the same subtree later on. Along with this contribution, we also proposed to use work stealing to deal with the irregularity problems of parallel B&B. Through extensive experimental study, we also confirmed that work stealing paradigm is a good candidate in this context. This observation in fact establishes a base for further investigations in order to improve the performance of parallel B&B on different computing systems.

In the second contribution, we mainly targeted the design of an efficient approach for parallel B&B on heterogeneous computing resources comprising of multiple CPUs and GPUs. The main objective is to ensure that all heterogeneous processing units run at their maximum capacity and an evenly balanced workload is maintained during execution. Firstly, we proposed the 2MBB approach in the context of completely distributed CPUs and GPUs. This approach deals with two-level parallelism allowing for (i) distributed subtree exploration among processing units and (ii) concurrent operations between every single GPU host and device. The 2MBB approach attains near-linear speedups when parallelizing B&B computations on relatively low and moderate distributed scales (up to 128 processing units). At

larger scales, the 2MBB however suffers from distributed communication cost and does not allow to fully use the power offered by the distributed system. Consequently, we proposed the 3MBB approach which extends the 2MBB for multi-core heterogeneous systems. The 3MBB approach deals with the hardware hierarchy (shared vs. distributed memory) in order to minimize communication cost. Within a multi-core system, decentralized split work pools are used to share B&B problems; and intra-node stealing operations are performed in order to acquire work efficiently while avoiding shared memory locking and synchronization issues. Our experimental results show up to 50% improvement compared to 2MBB in large scales.

In the last contribution, we proposed a Link-Heterogeneous Work Stealing (LWS) to deal with the link-heterogeneous issues when deploying parallel B&B on distributed geographical computing environments. The LWS is generic in the sense that it can be deployed in different computing contexts exposing different properties in terms of link latency. In LWS, an idle processing unit performs two types of stealing requests: one to locally preferred neighbors and another one to remote neighbors. This stealing mechanism is inspired by the CRS [Van Nieuwpoort 2001] and ACRS [Van Nieuwpoort 2004] algorithm which are specially designed for the grids of two-level hierarchy communication. However unlike the ACRS or CRS, the LWS can identify the preferred neighbors and remote neighbors during execution, hence resulting in the possibility of deploying it on several distributed computing environments. We performed a set of extensive experiments on different link-heterogeneous distributed environments, and the result confirmed that the LWS not only produces good performance on different types of computing environments, but also improves the performance of some state-of-the-art algorithms designed for specific computing environments.

As future research directions for this work, we have identified some challenging perspectives summarized in the following:

- In this thesis, we only consider to tackle either the node-heterogeneity or link-heterogeneity and relax the other one for the sake of simplicity. However the two issues always appear in real large scale distributed systems. Therefore, one direction to extend this work is to consider more realistic scenarios where node-heterogeneity and link-heterogeneity exist. A possible solution is to integrate the 3MBB and LWS into a comprehensive solution in order to efficiently deal with such complex distributed systems.

- Another research issue is to consider other hardwares components with new characteristics and to study at what extent load balancing can be managed efficiently. For instance, it would be interesting to consider new many-core shared memory systems with specific properties and to study how they can be taken into account within a complex distributed and heterogenous system.

- Load balancing is an important issue especially when the different levels of parallelism are hidden by the compute platform. This can be typically the case when considering virtualized and cloud environments. An open research

question is to study the feasibility of solving large scale COPs in such compute settings.

Bibliography

[Acar 2000] U.A. Acar, G.E. Blelloch and R.D. Blumofe. *The Data Locality of Work Stealing*. In 15^{th} ACM Symposium on Parallel Algorithms and Architectures (SPAA), pages 1–12, 2000. (Cited on page 86.)

[Aida 2002] K. Aida and Y. Futakata. *High-performance parallel and distributed computing for the BMI eigenvalue problem*. In Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM, pages 71–78, April 2002. (Cited on page 22.)

[Aida 2005] K. Aida and T. Osumi. *A case study in running a parallel branch and bound application on the grid*. In Applications and the Internet, 2005. Proceedings. The 2005 Symposium on, pages 164–173, Jan 2005. (Cited on page 22.)

[Apache] Apache. *Apache Hadoop. URL: http://hadoop.apache.org/*. (Cited on pages 15 and 20.)

[Baldeschwieler 1996] J.E. Baldeschwieler, R.D. Blumofe and E.A. Brewer. *ATLAS: an infrastructure for global computing*. In 7^{th} ACM SIGOPS Workshop on Systems support for worldwide applications, pages 165–172, 1996. (Cited on page 86.)

[Barreto 2010] Bauer M. Barreto L. *Parallel Branch and Bound Algorithm - A comparison between serial, OpenMP and MPI implementations*. Journal of Physics, vol. 256, no. 1, October 2010. (Cited on page 18.)

[Beaumont 2010a] O. Beaumont and AL. Rosenberg. *Link-heterogeneity vs. node-heterogeneity in clusters*. In High Performance Computing (HiPC), 2010 International Conference on, pages 1–8, Dec 2010. (Cited on page 14.)

[Beaumont 2010b] O. Beaumont and A.L. Rosenberg. *Link-heterogeneity vs. node-heterogeneity in clusters*. In International Conference on High Performance Computing (HiPC), pages 1–8, 2010. (Cited on page 88.)

[Bendjoudi 2012a] A. Bendjoudi. *Scalable and fault tolerant hierarchical B&B algorithms for Computational Grids*. Phd thesis, Algiers, Algeria, 2012. (Cited on pages 2, 3, 29, 34, 86 and 109.)

[Bendjoudi 2012b] A. Bendjoudi, N. Melab and E-G. Talbi. *An adaptive hierarchical master-worker framework for grids: Application to B&B algorithms*. J. Parallel Distrib. Comput (JPDC), vol. 72, no. 2, pages 120–131, 2012. (Cited on pages 22, 42, 43, 49 and 51.)

[Bendjoudi 2012c] A. Bendjoudi, N. Melab and E. G. Talbi. *Hierarchical Branch and Bound Algorithm for Computational Grids.* Future Gener. Comput. Syst., vol. 28, no. 8, pages 1168–1176, October 2012. (Cited on pages 22, 49 and 51.)

[Bittorrent] Bittorrent. *Bittorrent. URL: http://www.bittorrent.com/.* (Cited on page 15.)

[Blumofe 1999] R. D. Blumofe and C. E. Leiserson. *Scheduling multithreaded computations by work stealing.* J. ACM, vol. 46, pages 720–748, 1999. (Cited on pages 2, 33, 35, 62 and 66.)

[Bourbeau 2000] Benoît Bourbeau, Teodor Gabriel Crainic and Bernard Gendron. *Branch-and-bound Parallelization Strategies Applied to a Depot Location and Container Fleet Management Problem.* Parallel Comput., vol. 26, no. 1, pages 27–46, January 2000. (Cited on pages 8 and 9.)

[Budiu 2011] Mihai Budiu, Daniel Delling and Renato F. Werneck. *DryadOpt: Branch-and-Bound on Distributed Data-Parallel Execution Engines.* In Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS '11, pages 1278–1289, Washington, DC, USA, 2011. IEEE Computer Society. (Cited on page 20.)

[Burton 1981] F. Warren Burton and M. Ronan Sleep. *Executing Functional Programs on a Virtual Tree of Processors.* In Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture, FPCA '81, pages 187–194, New York, NY, USA, 1981. ACM. (Cited on pages 2, 30 and 32.)

[Cappello 2005] F. Cappello, P. Fraigniaud, B. Mans and A.L. Rosenberg. *An algorithmic model for heterogeneous hyper-clusters: rationale and experience.* Int. J. Found. Comput. Sci., vol. 16, no. 2, pages 195–215, 2005. (Cited on page 88.)

[Casado 2008] L. G. Casado, J. A. Martinez, I. Garcia and E. M. T. Hendrix. *Branch-and-Bound Interval Global Optimization on Shared Memory Multiprocessors.* Optimization Methods Software, vol. 23, no. 5, pages 689–701, October 2008. (Cited on pages 10 and 18.)

[Chakroun 2012] I. Chakroun and M. Melab. *An Adaptative Multi-GPU based Branch-and-Bound. A Case Study: the Flow-Shop Scheduling Problem.* In 14^{th} IEEE Inter. Conf. On High Performance Computing and Communications, 2012. (Cited on pages 61, 73 and 75.)

[Chakroun 2013a] I. Chakroun. *Parallel heterogeneous Branch and Bound algorithms for multi-core and multi-GPU environments.* Phd thesis, Lille, France, 2013. (Cited on pages 2 and 34.)

[Chakroun 2013b] I. Chakroun, N. Melab, M. Mezmaz and D. Tuyttens. *Combining Multi-core and GPU Computing for Solving Combinatorial Optimization Problems*. J. Parallel Distrib. Comput., vol. 73, no. 12, pages 1563–1577, December 2013. (Cited on page 69.)

[Chen] Ferris M. C. Chen Q. *FATCOP: A fault tolerant condor-PVM Mixed Integer Programming Solver*. Technical report, University of Wisconsin-Madison. (Cited on page 20.)

[Chiu 1989] Dah-Ming Chiu and Raj Jain. *Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks*. Comput. Netw. ISDN Syst., vol. 17, no. 1, pages 1–14, 1989. (Cited on page 97.)

[CoralCDN] CoralCDN. *Coral Content Distribution Network. URL: http://www.yacy.de/*. (Cited on page 15.)

[Dean 2008] Jeffrey Dean and Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. Commun. ACM, vol. 51, no. 1, pages 107–113, January 2008. (Cited on page 20.)

[Diconstanzo 2007] A. Diconstanzo. *Branch-and-Bound with peer-to-peer for large scale grids*. Phd thesis, Ecole doctorale STIC, France, 2007. (Cited on pages 22 and 23.)

[Dinan 2007] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan and C.-W. Tseng. *Dynamic Load Balancing of Unbalanced Computations Using Message Passing*. In 21^{th} IPDPS, pages 1–8, 2007. (Cited on page 33.)

[Dinan 2009] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy and Jarek Nieplocha. *Scalable Work Stealing*. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, pages 53:1–53:11, New York, NY, USA, 2009. ACM. (Cited on pages 33 and 70.)

[dis] *DISTributed systems EMulator*. http://distem.gforge.inria.fr/. (Cited on page 98.)

[Djamai 2011a] M. Djamai, B. Derbel and N. Melab. *Distributed B&B: A Pure Peer-to-Peer Approach*. In Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on, pages 1788–1797, May 2011. (Cited on pages 23 and 24.)

[Djamai 2011b] Mathieu Djamai, Bilel Derbel and Nouredine Melab. *Distributed B&B: A Pure Peer-to-Peer Approach*. In Workshop on Large-Scale Parallel Processing (LSPP'11), Anchorage, Alaska, USA, May 2011. (Cited on page 10.)

[Djamai 2013] M. Djamai. *Algorithmes Branch-and-Bound Pair-a-Pair pour grilles de calcul*. Phd thesis, Lille, France, 2013. (Cited on pages 2, 29, 34 and 86.)

[Dongarra 2006] J. Dongarra and A. Lastovetsky. *An Overview of Heterogeneous High Performance and Grid Computing*. Engineering the Grid: Status and Perspective, 2006. (Cited on page 86.)

[Drozdowski 2012] Maciej Drozdowski, Pawel Marciniak, Grzegorz Pawlak and Maciej Plaza. *Grid Branch-and-bound for Permutation Flowshop*. In Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics - Volume Part II, PPAM'11, pages 21–30, Berlin, Heidelberg, 2012. Springer-Verlag. (Cited on page 22.)

[Dunham 2002] M.H. Dunham. Data mining: introductory and advanced topics. Prentice Hall, 2002. (Cited on page 94.)

[Eager 1986] D.L. Eager, E.D. Lazowska and J. Zahorjan. *Adaptive load sharing in homogeneous distributed systems*. IEEE Transactions on Software Engineering,, vol. SE-12, no. 5, pages 662–675, May 1986. (Cited on pages 30 and 31.)

[Eckstein 2001] Jonathan Eckstein, Cynthia A. Phillips and William E. Hart. *PICO: An Object-Oriented Framework for Parallel Branch and Bound*. Technical report, Rutgers University, 2001. (Cited on page 22.)

[Evtushenko 2009] Yuri Evtushenko, Mikhail Posypkin and Israel Sigal. *A framework for parallel large-scale global optimization*. Computer Science - Research and Development, vol. 23, no. 3-4, pages 211–215, 2009. (Cited on page 19.)

[Fedak 2001] G. Fedak, C. Germain, V. Neri and F. Cappello. *XtremWeb: a generic global computing system*. In IEEE/ACM International Symposium on Cluster Computing and the Grid, pages 582–587, 2001. (Cited on page 86.)

[Finkel 1987] Raphael Finkel and Udi Manber. *DIB - a distributed implementation of backtracking*. ACM Trans. Program. Lang. Syst., vol. 9, pages 235–256, 1987. (Cited on page 23.)

[Frigo 1998] M. Frigo, E. Leiserson C. and H. Randal K. *The implementation of the Cilk-5 multithreaded language*. SIGPLAN Not., vol. 33, pages 212–223, 1998. (Cited on page 31.)

[Garey 1976] M. R. Garey, D. S. Johnson and R. Sethi. *The complexity of flowshop and jobshop scheduling*. Mathematics of Operations Research, vol. 1, pages 117–129, 1976. (Cited on page 41.)

[Garlaschelli 2009] D. Garlaschelli. *The weighted random graph model*. New Journal of Physics, vol. 11, no. 7, 2009. (Cited on pages 103 and 104.)

[Gast 2010] N. Gast and G. Bruno. *A Mean Field Model of Work Stealing in Large-scale Systems*. In ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), pages 13–24, 2010. (Cited on page 86.)

[Gendron 1994] B. Gendron and T.G. Crainic. *Parallel Branch-and-Bound Algorithms: Survey and Synthesis.* Operations Research, vol. 42, pages 1042–1066, 1994. (Cited on pages 8 and 9.)

[Goux 2000] J.-P. Goux, S. Kulkarni, J. Linderoth and M. Yoder. *An enabling framework for master-worker applications on the Computational Grid.* In High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on, pages 43–50, 2000. (Cited on page 20.)

[Grid'5000] Grid'5000. *Grid'5000 French national gird. URL: https://www.grid5000.fr/.* (Cited on pages 42, 73, 88 and 100.)

[Iamnitchi 2000] A Iamnitchi and I Foster. *A problem-specific fault-tolerance mechanism for asynchronous, distributed systems.* In Parallel Processing, 2000. Proceedings. 2000 International Conference on, pages 4–13, 2000. (Cited on page 23.)

[Intel] Intel. *Intel Threading Building Blocks.* (Cited on page 31.)

[Isard 2007] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell and Dennis Fetterly. *Dryad: Distributed Data-parallel Programs from Sequential Building Blocks.* SIGOPS Oper. Syst. Rev., vol. 41, no. 3, pages 59–72, March 2007. (Cited on page 20.)

[Janjic 2013] V. Janjic and K. Hammond. *How to be a Successful Thief.* In 19^{th} Inter. Conf. on Parallel Proc. (Euro-Par), pages 114–125, 2013. (Cited on page 86.)

[Janson 1998] S. Janson. *One, Two And Three Times log(n)/n For Paths In A Complete Graph With Random Weights.* Combin. Probab. Comp., vol. 8, pages 347–361, 1998. (Cited on page 103.)

[Kielmann 2002] T. Kielmann, H.E. Bal, J. Maassen, R. Van Nieuwpoort, L. Eyraud, R. Hofman and K. Verstoep. *Programming environments for high-performance Grid computing: the Albatross project.* Future Generation Computer Systems, vol. 18, no. 8, pages 1113 – 1125, 2002. (Cited on page 86.)

[Korpela 2001] E. Korpela, D. Werthimer, D. Anderson, J. Cobb and M. Leboisky. *SETI@home - massively distributed computing for SETI.* Computing in Science Engineering, vol. 3, no. 1, pages 78–83, 2001. (Cited on pages 20 and 86.)

[Kouki 2010] Samia Kouki, Mohamed Jemni and Talel Ladhari. *Deployment of Solving Permutation Flow Shop Scheduling Problem on the Grid.* In Taihoon Kim, StephenS. Yau, Osvaldo Gervasi, Byeong-Ho Kang, Adrian Stoica and Dominik Slezak, editeurs, Grid and Distributed Computing, Control and Automation, volume 121 of *Communications in Computer and Information Science*, pages 95–104. Springer Berlin Heidelberg, 2010. (Cited on page 21.)

[Kouki 2012] Samia Kouki, Mohamed Jemni and Talel Ladhari. *A Load Balanced Distributed Algorithm to Solve the Permutation Flow Shop Problem Using the Grid*. In Proceedings of the 2012 IEEE 15th International Conference on Computational Science and Engineering, CSE '12, pages 146–153, Washington, DC, USA, 2012. IEEE Computer Society. (Cited on page 21.)

[Kouki 2013] S. Kouki, M. Jemni and T. Ladhari. *Scalable Distributed Branch and Bound for the Permutation Flow Shop Problem*. In P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2013 Eighth International Conference on, pages 503–508, Oct 2013. (Cited on page 21.)

[Kumar 1994] Vipin Kumar, Ananth Y. Grama and Nageshwara Rao Vempaty. *Scalable Load Balancing Techniques for Parallel Computers*. J. Parallel Distrib. Comput., vol. 22, no. 1, pages 60–79, July 1994. (Cited on page 30.)

[Ladhari 2005] Talel Ladhari and Mohamed Haouari. *A Computational Study of the Permutation Flow Shop Problem Based on a Tight Lower Bound*. Comput. Oper. Res., vol. 32, no. 7, pages 1831–1847, July 2005. (Cited on page 21.)

[Lalami 2012] M. E. Lalami and D. El-Baz. *GPU Implementation of the Branch and Bound Method for Knapsack Problems*. In IPDPS Workshops, pages 1769–1777, 2012. (Cited on page 61.)

[Luling 1992] R. Luling and B. Monien. *Load balancing for distributed branch amp; bound algorithms*. In Parallel Processing Symposium, 1992. Proceedings., Sixth International, pages 543–548, Mar 1992. (Cited on page 23.)

[Maymounkov 2002] Petar Maymounkov and David Mazières. *Kademlia: A Peer-to-Peer Information System Based on the XOR Metric*. In Revised Papers from the First International Workshop on Peer-to-Peer Systems, IPTPS '01, pages 53–65, London, UK, UK, 2002. Springer-Verlag. (Cited on page 15.)

[Melab 2005] N. Melab. *Contribution a la resolution de problemes dóptimisation combinatoire sur grilles de calcul*. Habilitation a diriger des recherches de lúniversité lille1, Lille, France, 2005. (Cited on pages 8 and 9.)

[Melab 2012] N. Melab, I. Chakroun, M. Mezmaz and D. Tuyttens. *A GPU-accelerated B&B Algorithm for the Flow-Shop Scheduling Problem*. In 14^{th} IEEE Conf. on Cluster Computing, 2012. (Cited on pages 73 and 75.)

[Mezmaz 2007a] M. Mezmaz. *Une approche efficace pour le passage sur grilles de calcul de methodes dóptimisation combinatoire*. Phd thesis, Lille, France, 2007. (Cited on pages 2, 3, 9, 10, 29, 34, 86 and 109.)

[Mezmaz 2007b] M. Mezmaz, N. Melab and E-G. Talbi. *A Grid-based Parallel Approach of the Multi-Objective Branch and Bound*. In Proceedings of the 15th

Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP '07, pages 23–30, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on page 20.)

[Mezmaz 2007c] M.-S. Mezmaz, N. Melab and E.-G. Talbi. *A Grid-enabled Branch and Bound Algorithm for Solving Challenging Combinatorial Optimization Problems*. In Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, pages 1–9, March 2007. (Cited on pages 20, 42, 52 and 53.)

[Mezmaz 2013] Melab N. Tuyttens D. Mezmaz M. Large scale network-centric distributed systems, chapitre A Multithreaded branch-and-bound algorithm for solving the flow-shop problem on a multicore environment. John Wiley and Sons, USA, 2013. (Cited on pages 19 and 69.)

[Min 2011] Seung-Jai Min, Costin Iancu and Katherine Yelick. *Hierarchical Work Stealing on Manycore Clusters*. In 5^{th} Conf. on Partitioned Global Address Space Prog. Models, Oct 2011. (Cited on pages 35 and 66.)

[Napster] Napster. *Napster. URL: http://www.napster.com/*. (Cited on page 15.)

[Neary 2000] MichaelO. Neary, Alan Phipps, Steven Richman and Peter Cappello. *Javelin 2.0: Java-Based Parallel Computing on the Internet*. In Euro-Par 2000 Parallel Processing, volume 1900 of *Lecture Notes in Computer Science*, pages 1231–1238. Springer Berlin Heidelberg, 2000. (Cited on page 31.)

[Olivier 2007] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan and Chau-Wen Tseng. *UTS: an unbalanced tree search benchmark*. In 19^{th} inter. conf. on Languages and compilers for parallel computing (LCPC), pages 235–250, 2007. (Cited on pages 41 and 47.)

[Olivier 2008] Stephen Olivier and Jan Prins. *Scalable Dynamic Load Balancing Using UPC*. In 37^{th} International Conference on Parallel Processing (ICPP), pages 123–131, 2008. (Cited on page 47.)

[OpenMP 2011] OpenMP. *OpenMP Application Program Interface*, 2011. (Cited on page 31.)

[Otten] Dechter R. Otten L. *Load Balancing for Parallel Branch and Bound*. Technical report, University of California Irvine. (Cited on page 20.)

[Peleg 1989] D. Peleg and A.A. Schäffer. *Graph spanners*. Journal of Graph Theory, vol. 13, no. 1, pages 99–116, 1989. (Cited on page 104.)

[Pilla 2012] L.L. Pilla, C.P. Ribeiro, D. Cordeiro, Chao Mei, A. Bhatele, P.O.A. Navaux, F. Broquedis, J. Mehaut and L.V. Kale. *A Hierarchical Approach for Load Balancing on Parallel Multi-core Systems*. In 41^{st} Inter. Conf. on Parallel Processing (ICPP), pages 118–127, 2012. (Cited on pages 86 and 87.)

[Plaat 1999] A. Plaat, H.E. Bal and R. F H Hofman. *Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects.* In HPCA, pages 244–253, 1999. (Cited on page 86.)

[Quintin 2010] J.-N. Quintin and F. Wagner. *Hierarchical Work-stealing.* In 16^{th} Inter. Conf. on Parallel Processing (EuroPar), pages 217–229, 2010. (Cited on page 89.)

[Sarzyniec 2013] L. Sarzyniec, T. Buchert, E. Jeanvoine and L. Nussbaum. *Design and Evaluation of a Virtual Experimental Environment for Distributed Systems.* In 21^{st} Inter. Conf. on Parallel, Distributed and Network-Based Processing, 2013. (Cited on page 98.)

[Savadi 2012] Abdorreza Savadi and Hossein Deldari. *A Bridging Model for Branch-and-Bound Algorithms on Multi-core Architectures.* In Proceedings of the 2012 Fifth International Symposium on Parallel Architectures, Algorithms and Programming, PAAP '12, pages 235–241, Washington, DC, USA, 2012. IEEE Computer Society. (Cited on page 19.)

[Shivaratri 1992a] Niranjan G. Shivaratri, Phillip Krueger and Mukesh Singhal. *Load Distributing for Locally Distributed Systems.* Computer, vol. 25, no. 12, pages 33–44, December 1992. (Cited on pages 30 and 31.)

[Shivaratri 1992b] Niranjan G. Shivaratri, Phillip Krueger and Mukesh Singhal. *Load Distributing for Locally Distributed Systems.* Computer, vol. 25, no. 12, pages 33–44, December 1992. (Cited on page 31.)

[Silva 2014] Boeres C. Drummond L. M. A. Silva J. M. N. and A. A. Pessoa. *Memory aware load balance strategy on a parallel branch-and-bound application.* Concurrency and Computation: Practice and Experience, 2014. (Cited on page 19.)

[Stoica 2001] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek and Hari Balakrishnan. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.* SIGCOMM Comput. Commun. Rev., vol. 31, no. 4, pages 149–160, August 2001. (Cited on page 15.)

[Taillard 1993] E. Taillard. *Benchmarks for basic scheduling problems.* European Journal of Operational Research, vol. 64, no. 2, pages 278–285, 1993. (Cited on page 41.)

[Top500] Top500. *Top500 SuperComputers. URL: https://www.top500.com/.* (Cited on page 1.)

[Trienekens 1986] H.W.J.M. Trienekens. *Parallel Branch and Bound on an MIMD System.* Technical report, Econometric Institute, Erasmus University, Rotterdam, Netherlands, 1986. (Cited on page 8.)

[Trienekens 1992] Bruin A. Trienekens H.W.J.M. *Towards a Taxonomy of Parallel Branch and Bound Algorithms*. Technical report, Econometric Institute, Erasmus University, Rotterdam, Netherlands, 1992. (Cited on page 8.)

[Tschoke 1995] S. Tschoke, R. Lubling and B. Monien. *Solving the traveling salesman problem with a distributed branch-and-bound algorithm on a 1024 processor network*. In Parallel Processing Symposium, 1995. Proceedings., 9th International, pages 182–189, Apr 1995. (Cited on page 23.)

[Van Nieuwpoort 2001] R. Van Nieuwpoort, T. Kielmann and H.E. Bal. *Efficient Load Balancing for Wide-area Divide-and-conquer Applications*. In 8^{th} Symp. on Principles and Practices of Para. Programming, pages 34–43, 2001. (Cited on pages 86, 90, 91 and 110.)

[Van Nieuwpoort 2004] R. Van Nieuwpoort, J. Maassen, G. Wrzesinska, T. Kielmann and H. E. Bal. *Adaptive Load-Balancing for Divide-and-Conquer Grid Applications*. Journal of Supercomputing, 2004. (Cited on pages 86, 93 and 110.)

[Vu 2012] Trong-Tuan Vu, Bilel Derbel, Ali Asim, Ahcene Bendjoudi and Nouredine Melab. *Overlay-Centric Load Balancing: Applications to UTS and B&B*. In 14^{th} IEEE International Conference on Cluster Computing (CLUSTER), pages 382–390, 2012. (Cited on page i.)

[Vu 2013] Trong-Tuan Vu, Bilel Derbel and Nouredine Melab. *Adaptive Dynamic Load Balancing in Heterogeneous Multiple GPUs-CPUs Distributed Setting: Case Study of B&B Tree Search*. In 7^{th} Learning and Intelligent Optimization (LION), pages 87–103. Springer Berlin Heidelberg, 2013. (Cited on page i.)

[Vu 2014a] Trong-Tuan Vu and B. Derbel. *Link-Heterogeneous Work Stealing*. In 14^{th} IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pages 354–363, May 2014. (Cited on page ii.)

[Vu 2014b] Trong-Tuan Vu and Bilel Derbel. *Parallel Branch-and-Bound in Multi-core Multi-CPU Multi-GPU Heterogeneous Environments*. Research report, Sep 2014. (Cited on page i.)

[Xu 2005] Y. Xu, T. K. Ralphs, L. Ladányi and M. J. Saltzman. *ALPS: A framework for implementing parallel search algorithms*. In In Proceedings of the Ninth INFORMS Computing Society Conference, pages 319–334, 2005. (Cited on page 22.)

[Yacy] Yacy. *Yacy, URL: http://www.yacy.de/*. (Cited on page 15.)

[Zhai 2011] Y. Zhai, M. Liu, J. Zhai, X. Ma and W. Chen. *Cloud versus in-house cluster: Evaluating Amazon cluster compute instances for running MPI applications*. In Inter. Conference for High Performance Computing, Networking, Storage and Analysis (SC), pages 1–10, 2011. (Cited on pages 86 and 87.)